Low-Level Software Security for Compiler Developers

Version 0-80-g26f97d3

Contents

Chapter 1

Introduction

Compilers, assemblers and similar tools generate all the binary code that processors execute. It is no surprise then that for security analysis and hardening relevant for binary code, these tools have a major role to play. Often the only practical way to protect all binaries with a particular security hardening method is to let the compiler adapt its automatic code generation.

With software security becoming even more important in recent years, it is no surprise to see an ever increasing variety of security hardening features and mitigations against vulnerabilities implemented in compilers.

Indeed, compared to a few decades ago, today's compiler developer is much more likely to work on security features, at least some of their time.

Furthermore, with the ever-expanding range of techniques implemented, it has become very hard to gain a basic understanding of all security features implemented in typical compilers.

This poses a practical problem: compiler developers must be able to work on security hardening features, yet it is hard to gain a good basic understanding of such compiler features.

This book aims to help developers of code generation tools such as JITs, compilers, linkers and assemblers to overcome this.

There is a lot of material that can be found explaining individual vulnerabilities or attack vectors. There are also lots of presentations explaining specific exploits. But there seems to be a limited set of material that gives a structured overview of all vulnerabilities and exploits for which a code generator could play a role in protecting against them.

This book aims to provide such a structured, broad overview. It does not necessarily go into full details. Instead it aims to give a thorough description of all relevant high-level aspects of attacks, vulnerabilities, mitigations and hardening techniques. For further details, this book provides pointers to material with more details on specific techniques.

The purpose of this book is to serve as a guide to every compiler developer that

needs to learn about software security relevant to compilers. Even though the focus is on compiler developers, we expect that this book will also be useful to other people working on low-level software.

1.1 Why an open source book?

The idea for this book emerged out of a frustration of not finding a good overview on this topic. Kristof Beyls and Georgia Kouveli, both compiler engineers working on security features, wished a book like this would exist. After not finding such a book, they decided to try and write one themselves. They immediately realized that they do not have all necessary expertise themselves to complete such a daunting task. So they decided to try and create this book in an open source style, seeking contributions from many experts.

As you read this, the book remains unfinished. This book may well never be finished, as new vulnerabilities continue to be discovered regularly. Our hope is that developing the book as an open source project will allow for it to continue to evolve and improve. The open source development process of this book increases the likelihood that it remains relevant as new vulnerabilities and mitigations emerge.

Kristof and Georgia, the initial authors, are far from experts on all possible vulnerabilities. So what is the plan to get high quality content to cover all relevant topics? It is two-fold.

First, by studying specific topics, they hope to gain enough knowledge to write up a good summary for this book.

Second, they very much invite and welcome contributions. If you're interested in potentially contributing content, please go to the home location for the open source project at https://github.com/llsoftsec/llsoftsecbook.

As a reader, you can also contribute to making this book better. We highly encourage feedback, both positive and constructive criticisms. We prefer feedback to be received through https://github.com/llsoftsec/llsoftsecbook.

Add section describing the structure of the rest of the book.

Chapter 2

Memory vulnerability based attacks and mitigations

2.1 A bit of background on memory vulnerabilities

Memory access errors describe memory accesses that, although permitted by a program, were not intended by the programmer. These types of errors are usually defined (Hicks 2014) by explicitly listing their types, which include:

- buffer overflow
- null pointer dereference
- use after free
- use of uninitialized memory
- illegal free

Memory vulnerabilities are an important class of vulnerabilities that arise due to these types of errors, and they most commonly occur due to programming mistakes when using languages such as C/C++. These languages do not provide mechanisms to protect against memory access errors by default. An attacker can exploit such vulnerabilities to leak sensitive data or overwrite critical memory locations and gain control of the vulnerable program.

Memory vulnerabilities have a long history. The Morris worm in 1988 was the first widely publicized attack exploiting a buffer overflow. Later, in the mid-90s, a few famous write-ups describing buffer overflows appeared (Aleph One 1996). Stack buffer overflows were mitigated with stack canaries and non-executable stacks. The answer was more ingenious ways to bypass these mitigations: code reuse attacks, starting with attacks like return-into-libc (Solar Designer 1997). Code reuse attacks later evolved to Return-Oriented Programming (ROP) (Shacham 2007) and even more complex techniques.

To defend against code reuse attacks, the Address Space Layout Randomization (ASLR) and Control-Flow Integrity (CFI) measures were introduced. This interaction between offensive and defensive security research has been essential

to improving security, and continues to this day. Each newly deployed mitigation results in attempts, often successful, to bypass it, or in alternative, more complex exploitation techniques, and even tools to automate them.

Memory safe (Hicks 2014) languages are designed with prevention of such vulnerabilities in mind and use techniques such as bounds checking and automatic memory management. If these languages promise to eliminate memory vulnerabilities, why are we still discussing this topic?

On the one hand, C and C++ remain very popular languages, particular in the implementation of low-level software. On the other hand, programs written in memory safe languages can themselves be vulnerable to memory errors as a result of bugs in how they are implemented, e.g. a bug in their compiler. Can we fix the problem by also using memory safe languages for the compiler and runtime implementation? Even if that were as simple as it sounds, unfortunately there are types of programming errors that these languages cannot protect against. For example, a logical error in the implementation of a compiler or runtime for a memory safe language can lead to a memory access error not being detected. We will see examples of such logic errors in compiler optimizations in a later section.

Given the rich history of memory vulnerabilities and mitigations and the active developments in this area, compiler developers are likely to encounter some of these issues over the course of their careers. This chapter aims to serve as an introduction to this area. We start with a discussion of exploitation primitives, which can be useful when analyzing threat models . We then continue with a more detailed discussion of the various types of vulnerabilities, along with their mitigations, presented in a rough chronological order of their appearance, and, therefore, complexity.

Discuss threat models elsewhere in book and refer to that section here

2.2 Exploitation primitives

Newcomers to the area of software security may find themselves lost in many blog posts and other publications describing specific memory vulnerabilities and how to exploit them. Two very common, yet unfamiliar to a newcomer, terms that appear in such publications are *read primitive* and *write primitive*. In order to understand memory vulnerabilities and be able to design effective mitigations, it's important to understand what these terms mean, how these primitives could be obtained by an attacker, and how they can be used.

An *exploit primitive* is a mechanism that allows an attacker to perform a specific operation in the memory space of the victim program. This is done by providing specially crafted input to the victim program.

A *write primitive* gives the attacker some level of write access to the victim's memory space. The value written and the address written to may be controlled by the attacker to various degrees. The primitive, for example, may allow:

- writing a fixed value to an attacker-controlled address, or
- writing to an address consisting of a fixed base and an attacker-controlled offset limited to a specific range (e.g. a 32-bit offset) , or
- writing to an attacker-controlled base address with a fixed offset.

Consider describing in more detail why the range limitation matters

Primitives can be further classified according to more detailed properties. See slide 11 of (Miller, n.d.) for an example.

The most powerful version of a write primitive is an *arbitrary write* primitive, where both the address and the value are fully controlled by the attacker.

A *read primitive*, respectively, gives the attacker read access to the victim's memory space. The address of the memory location accessed will be controlled by the attacker to some degree, as for the write primitive. A particularly useful primitive is an *arbitrary read* primitive, in which the address is fully controlled by the attacker.

The effects of a write primitive are perhaps easier to understand, as it has obvious side-effects: a value is written to the victim program's memory. But how can an attacker observe the result of a read primitive?

This depends on whether the attack is interactive or non-interactive (Hu et al. 2016).

- In an *interactive attack*, the attacker gives malicious input to the victim program. The malicious input causes the victim program to perform the read the attacker instructed it to, and to output the results of that read. This output could be any kind of output, for example a network packet that the victim transmits. The attacker can observe the result of the read primitive by looking at this output, for example parsing this network packet. This process then repeats: the attacker sends more malicious input to the victim, observes the output and prepares the next input. You can see an example of this type of attack in (Beer 2020), which describes a zero-click radio proximity exploit.
- In a *non-interactive (one-shot) attack*, the attacker provides all malicious input to the victim program at once. The malicious input triggers multiple primitives one after the other, and the primitives are able to observe the effects of the preceding operations through the victim program's state. The input could be, for example, in the form of a JavaScript program (Groß 2020), or a PDF file pretending to be a GIF (Beer and Groß 2021).

How does an attacker obtain these kinds of primitives in the first place? The details vary, and in some cases it takes a combination of many techniques, some of which are out of scope for this book. But we will be describing a few of them in this chapter. For example a stack buffer overflow results in a (restricted) write primitive when the input size exceeds what the program expected.

As part of an attack, the attacker will want to execute each primitive more than once, since a single read or write operation will rarely be enough to achieve their end goal (more on this later). How can primitives be combined to perform multiple reads/writes?

In the case of an interactive attack, preparing and sending input to the victim program and parsing the output of the victim program are usually done in an external program that drives the exploit. The attacker is free to use a programming language of their choice, as long as they can interact with the victim program in it. Let's assume, for example, an exploit program in C, communicating with the victim program over TCP. In this case, the primitives are abstracted into C functions, which prepare and send packets to the victim,

The references in this section describe complicated modern exploits. Consider linking to simpler exploits, as well as some tutorial-level material.

and parse the victim's responses. Using the primitives is then as simple as calling these functions. These calls can be easily combined with arbitrary computations, all written in C, to form the exploit.

For this cycle of repeated input/output interactions to work, the state of the victim program must not be lost between the different iterations of providing input and observing output. In other words, the victim process must not be restarted.

It's interesting to note that while the read/write primitives consist of carefully constructed inputs to the victim program, the attacker can view these inputs as *instructions* to the victim program. The victim program effectively implements an interpreter unintentionally, and the attacker can send instructions to this interpreter. This is explored further in (Dullien 2020).

In the case of a non-interactive attack, all computation happens within the victim program. The duality of input data and code is even more obvious in this case, as the malicious input to the victim can be viewed as the exploit code. There are cases for which the input is obviously interpreted as code by the victim application as well, as in the case of a JavaScript program given as input to a JavaScript engine. In this case, the read/write primitives would be written as JavaScript functions, which when called have the unintended side-effect of accessing arbitrary memory that a JavaScript program is not supposed to have access to. The primitives can be chained together with arbitrary computations, also expressed in JavaScript.

There are, however, cases where the correspondence between data and code isn't as obvious. For example, in (Beer and Groß 2021), the malicious input consists of a PDF file, masquerading as a GIF. Due to an integer overflow bug in the PDF decoder, the malicious input leads to an unbounded buffer access, therefore to an arbitrary read/write primitive. In the case of JavaScript engine exploitation, the attacker would normally be able to use JavaScript operations and perform arbitrary computations, making exploitation more straightforward. In this case, there are no scripting capabilities officially supported. The attackers, however, take advantage of the compression format intricacies to implement a small computer architecture, in thousands of simple commands to the decoder. In this way, they effectively *introduce* scripting capabilities and are able to express their exploit as a program to this architecture.

So far, we have described read/write primitives. We have also discussed how an attacker might perform arbitrary computations:

- in an external program in the case of interactive attacks, or
- by using scripting capabilities (whether originally supported or introduced by the attacker) in non-interactive attacks.

Assuming an attacker has gained these capabilities, how can they use them to achieve their goals?

The ultimate goal of an attacker may vary: it may be, among other things, getting access to a system, leaking sensitive information or bringing down a service. Frequently, a first step towards these wider goals is arbitrary code execution within the victim process. We have already mentioned that the attacker will typically have arbitrary computation capabilities at this point, but arbitrary

code execution also involves things like calling arbitrary library functions and performing system calls.

Some examples of how the attacker may use the obtained primitives:

- Leak information, such as pointers to specific data structures or code, or the stack pointer.
- Overwrite the stack contents, e.g. to perform a ROP attack.
- Overwrite non-control data, e.g. authorization state. Sometimes this step is sufficient to achieve the attacker's goal, bypassing the need for arbitrary code execution.

Once arbitrary code execution is achieved, the attacker may need to exploit additional vulnerabilities in order to escape a process sandbox, escalate privilege, etc. Such vulnerability chaining is common, but for the purposes of this chapter we will focus on:

- Preventing memory vulnerabilities in the first place, thus stopping the attacker from obtaining powerful read/write primitives.
- Mitigating the effects of read/write primitives, e.g. with mechanisms to maintain Control-Flow Integrity (CFI).

2.3 Stack buffer overflows

A buffer overflow occurs when a read from or write to a data buffer exceeds its boundaries. This typically results in adjacent data structures being accessed, which has the potential of leaking or compromising the integrity of this adjacent data.

When the buffer is allocated on the stack, we refer to a stack buffer overflow. In this section we focus on stack buffer overflows since, in the absence of any mitigations, they are some of the simplest buffer overflows to exploit.

The stack frame of a function includes important control information, such as the saved return address and the saved frame pointer. Overwriting these values unintentionally will typically result in a crash, but the overflowing values can be carefully chosen by an attacker to gain control of the program's execution.

Here is a simple example of a program vulnerable to a stack buffer overflow[1]:

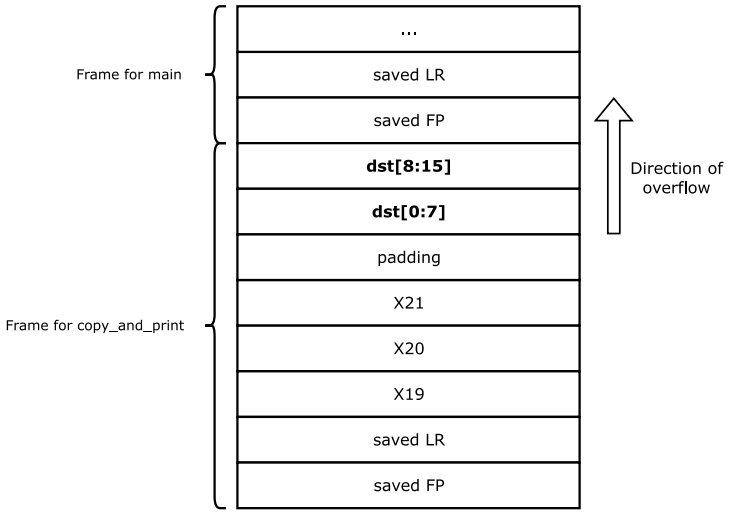
```
#include <stdio.h>
#include <string.h>

void copy_and_print(char* src) {
  char dst[16];

  for (int i = 0; i < strlen(src) + 1; ++i)
    dst[i] = src[i];
  printf("%s\n", dst);
}
```

[1]This is an oversimplified example for illustrative purposes. However, as this is a wide class of vulnerabilities, many real-world examples can be found and studied.

```
int main(int argc, char* argv[]) {
  if (argc > 1) {
    copy_and_print(argv[1]);
  }
}
```

In the code above, since the length of the argument is not checked before copying it into `dst`, we have a potential for a buffer overflow.

When looking at code generated for AArch64 with GCC 11.2[2], the stack layout looks like this:



Stack frame layout for stack buffer overflow example

The exact details of the stack frame layout, including the ordering of variables and the exact control information stored, will depend on the specific compiler version you use and the architecture you compile for.

As can be seen the stack diagram, an overflowing write in function `copy_and_print` can overwrite the saved frame pointer (FP) and link register (LR) in `main`'s frame. When `copy_and_print` returns, execution continues in `main`. When `main` returns, however, execution continues from the address stored in the saved LR, which has been overwritten. Therefore, when an attacker can choose the value that overwrites the saved LR, it's possible to control where the program resumes execution after returning from `main`.

Before non-executable stacks were mainstream, a common way to exploit these vulnerabilities would be to use the overflow to simultaneously write shellcode[3] to the stack and overwrite the return address so that it points to the shellcode. (Aleph One 1996) is a classic example of this technique.

[2]The code is generated with the `-fno-stack-protector` option, to ensure GCC's stack guard feature is disabled. We also used the `-O1` optimization level.

[3]A shellcode is a short instruction sequence that performs an action such as starting a shell on the victim machine.

The obvious solution to this issue is to use memory protection features of the processor in order to mark the stack (along with other data sections) as non-executable[4]. However, even when the stack is not executable, more advanced techniques can be used to exploit an overflow that overwrites the return address. These take advantage of code that already exists in the executable or in library code, and will be described in the next section.

Stack canaries are an alternative mitigation for stack buffer overflows. The general idea is to store a known value, called the stack canary, between the buffer and the control information (in the example, the saved FP and LR), and to check this value before leaving the function. Since an overflow that would overwrite the return address is going to overwrite the canary first, a corruption of the return address through a stack buffer overflow will be detected.

This technique has a few limitations: first of all, it specifically aims to protect against stack buffer overflows, and does nothing to protect against stronger primitives (e.g. arbitrary write primitives). Control-flow integrity techniques, which are described in the next section, aim to protect the integrity of stored code pointers against any modification.

Secondly, since a compiler needs to generate additional instructions for ensuring the canary's integrity, heuristics are usually employed to determine which functions are considered vulnerable. The additional instructions are then generated only for the functions that are considered vulnerable. Since heuristics aren't always perfect, this poses another potential limitation of the technique. To address this, compilers can introduce various levels of heuristics, ranging from applying the mitigations only to a small proportion of functions, to applying it universally. See, for example, the `-fstack-protector`, `-fstack-protector-strong` and `-fstack-protector-all` options offered by both GCC and Clang.

Another limitation is the possibility of leaks of the canary value. The canary value is often randomized at program start but remains the same during the program's execution. An attacker who manages to obtain the canary value at some point might, therefore, be able to reuse the leaked canary value and corrupt control information while avoiding detection. Choosing a canary value that includes a null byte (the C-style string terminator) might help in limiting the damage of overflows coming from string manipulation functions, even when the value is leaked.

Many buffer overflow vulnerabilities result from the use of unsafe library functions, such as `gets`, or from the unsafe use of library functions such as `strcpy`. There is extensive literature on writing secure C/C++ code, for example (Seacord 2013) and (Dowd, McDonald, and Schuh 2006). A different approach to limiting the effects of overflows is library function hardening, which aims to detect buffer overflows and terminate the program gracefully. This involves the introduction of feature macros like `_FORTIFY_SOURCE` (Sharma 2014).

Finally, it's important to mention that not all buffer overflows aim to overwrite a saved return address. There are many cases where a buffer overflow can overwrite other data adjacent to the buffer, for example an adjacent variable

[4]Note that the use of nested functions in GCC requires trampolines which reside on an executable stack. The use of nested functions, therefore, poses a security risk.

that determines whether authorization was successful, or a function pointer that, when modified, can modify the program's control flow according to the attacker's wishes.

Some of these vulnerabilities can be mitigated with the measures described in this section, but often more general measures to ensure memory safety or Control-Flow Integrity are necessary. For example, in addition to the hardening of specific library functions, compilers can also implement automatic bounds checking for arrays where the array bound can be statically determined (`-fsanitize=bounds`), as well as various other "sanitizers". We will describe these measures in following sections.


Rethink the later sections of this chapter and the order in which issues and mitigations are presented. The "non-control data exploits" section should probably include, or be followed by, a section on the various sanitizers available (ASan, UBSan, etc).

2.4 Code reuse attacks

In the early days of memory vulnerability exploitation, attackers could simply place shellcode of their choice in executable memory and jump to it. As non-executable stack and heap became mainstream, attackers started to reuse code already present in an application's binary and linked libraries instead. A variety of different techniques to this effect came to light.

The simplest of these techniques is return-to-libc (Solar Designer 1997). Instead of returning to shellcode that the attacker has injected, the return address is modified to return into a library function, such as `system` or `exec`. This technique is simpler to use when arguments are also passed on the stack and can therefore be controlled with the same stack buffer overflow that is used to modify the address.

2.4.1 Return-oriented programming

Return-to-libc attacks restrict an attacker to whole library functions. While this can lead to powerful attacks, it has also been demonstrated that it is possible to achieve arbitrary computation by combining a number of short instruction sequences ending in indirect control transfer instructions, known as **gadgets**. The indirect control transfer instructions make it easy for an attacker to execute gadgets one after another, by controlling the memory or register that provides each control transfer instruction's target.

In return-oriented programming (ROP) (Shacham 2007), each gadget performs a simple operation, for example setting a register, then pops a return address from the stack and returns to it. The attacker constructs a fake call stack (often called a ROP chain) which ensures a number of gadgets are executed one after another, in order to perform a more complex operation.

This will hopefully become more clear with an example: a ROP chain for AArch64 Linux that starts a shell, by calling `execve` with `"/bin/sh"` as an argument. The prototype of the `execve` library function, which wraps the exec system call, is:

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

For AArch64, `pathname` will be passed in the `x0` register, `argv` will be passed in `x1`, and `envp` in `x2`. For starting a shell, it is sufficient to:

- Make `x0` contain a pointer to `"/bin/sh"`.
- Make `x1` contain a pointer to an array of pointers with two elements:
  - The first element is a pointer to `"/bin/sh"`.
  - The second element is zero (`NULL`).
- Make `x2` contain zero (`NULL`).

This can be achieved by chaining gadgets to set the registers `x0`, `x1`, `x2`, and then returning to `execve` in the C library.

Let's assume we have the following gadgets:

1. A gadget that loads `x0` and `x1` from the stack:

```
gadget_x0_x1:
  ldp x0, x1, [sp]
  ldp x20, x19, [sp, #64]
  ldp x29, x30, [sp, #32]
  ldr x21, [sp, #48]
  add sp, sp, #0x50
  ret
```

2. A gadget that sets `x2` to zero, but also clears `x0` as a side-effect:

```
gadget_x2:
  mov x2, xzr
  mov x0, x2
  ldp x20, x19, [sp, #32]
  ldp x29, x30, [sp]
  ldr x21, [sp, #16]
  add sp, sp, #0x30
  ret
```


Explain how these gadgets could result from C/C++ code. The current versions are slightly tweaked by hand to have more manageable offsets.

Both gadgets also clobber several uninteresting registers, but since `gadget_x2` also clears `x0`, it becomes clear that we should use a ROP chain that:

1. Returns to `gadget_x2`, which sets `x2` to zero.
2. Returns to `gadget_x0_x1`, which sets `x0` and `x1` to the desired values.
3. Returns to `execve`.

Figure 2.1 shows this control flow.

We can achieve this by constructing the fake call stack shown in figure 2.2, where "Original frame" marks the frame in which the address of `gadget_x2` has replaced a saved return address that will be loaded and returned to in the future. As an alternative, an attacker could place this fake call stack somewhere else, for example on the heap, and use a primitive that changes the stack pointer's value instead. This is known as stack pivoting.

Note that this fake call stack contains NULL bytes, even without considering the exact values of the various return addresses included. An overflow bug that is based on a C-style string operation would not allow an attacker to replace the stack contents with this fake call stack in one go, since C-style strings are null-terminated and copying the fake stack contents would stop once the first

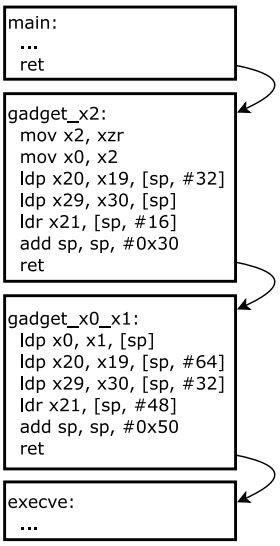


Figure 2.1: ROP example control flow

NULL byte is encountered. The ROP chain would therefore need to be adjusted so that it doesn't contain NULL bytes, for example by initially replacing the NULL bytes with a different byte and adding some more gadgets to the ROP chain that write zero to those stack locations.

A question that comes up when looking at the stack diagram is "how do we know the addresses of these gadgets"? We will talk a bit more about this in the next section.

ROP gadgets like the ones used here may be easy to identify by visual inspection of a disassembled binary, but it's common for attackers to use "gadget scanner" tools in order to discover large numbers of gadgets automatically. Such tools can also be useful to a compiler engineer working on a code reuse attack mitigation, as they can point out code sequences that should be protected and have been missed.

2.4.2 Jump-oriented programming

Jump-oriented programming (JOP) (Bletsch et al. 2011) is a variation on ROP, where gadgets can also end in indirect branch instructions instead of return instructions. The attacker chains a number of such gadgets through a dispatcher gadget, which loads pointers one after another from an array of pointers, and branches to each one in return. The gadgets used must be set up so that they branch or return back to the dispatcher after they're done. This is demonstrated in figure 2.3.

In figure 2.3, `x4` initially points to the "dispatch table", which has been modified by the attacker to contain the addresses of the three gadgets they want to execute. The dispatcher gadget loads each address in the dispatch table one by one and branches to them. The first gadget loads `x0` and `x1` from the stack, where the attacker has placed the inputs of their choice. It then loads its return address,

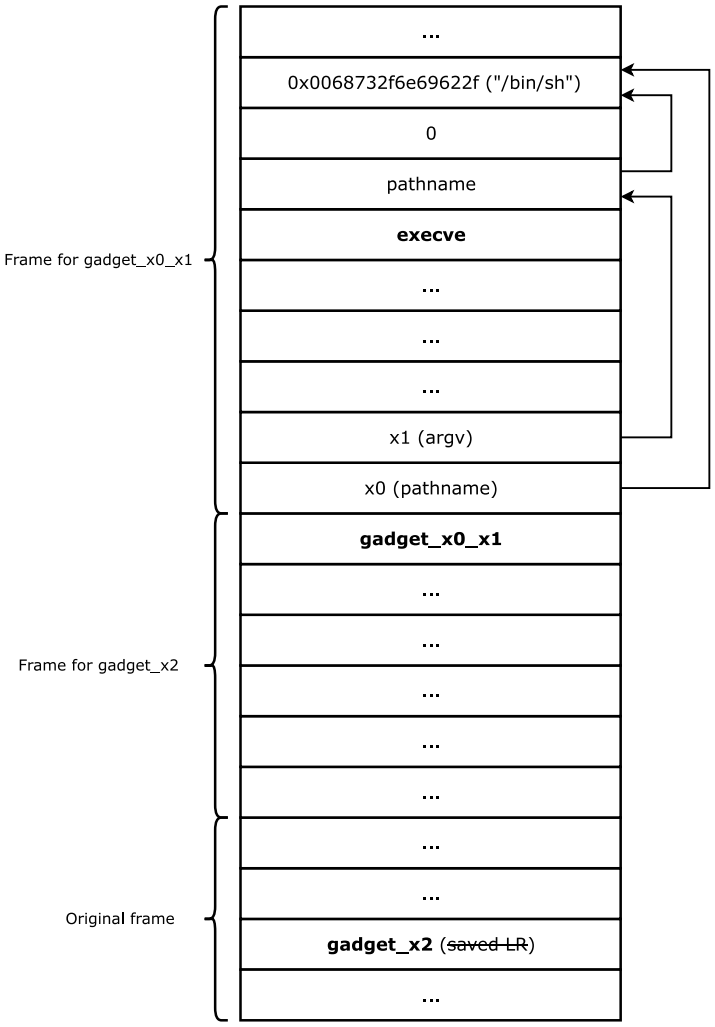


Figure 2.2: ROP example fake call stack

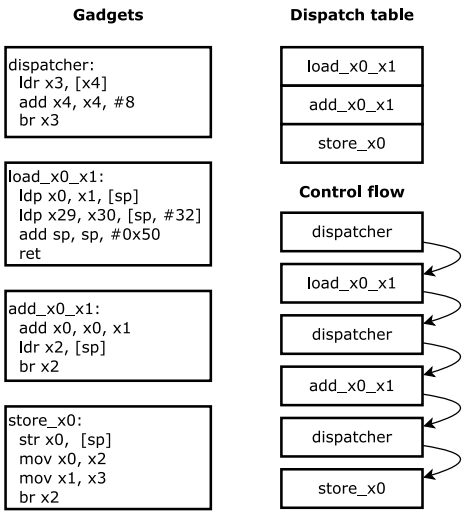


Figure 2.3: JOP example

also modified by the attacker so that it points back to the dispatcher gadget, and returns to it. The dispatcher branches to the next gadget, which adds `x0` and `x1` and leaves the result in `x0`, branching back to the dispatcher through another value loaded from the stack into `x2`. The final gadget stores the result of the addition, which remains in `x0`, to the stack, before branching to `x2`, which still points to the dispatcher gadget.

The gadgets in the figure are made up, chosen to highlight that each gadget can end in a different type of indirect control flow transfer instruction. Consider replacing them with more realistic ones.

2.4.3 Counterfeit Object-oriented programming

Counterfeit Object-oriented programming (COOP) (Schuster et al. 2015) is a code reuse technique that takes advantage of C++ virtual function calls. A COOP attack takes advantage of existing virtual functions and vtables, and creates fake objects pointing to these existing vtables. The virtual functions used as gadgets in the attack are called vfgadgets. To chain vfgadgets together, the attacker uses a "main loop gadget", similar to JOP's dispatcher gadget, which is itself a virtual function that loops over a container of pointers to C++ objects and invokes a virtual function on these objects. (Schuster et al. 2015) describes the attack in more detail. It is specifically mentioned here as an example of an attack that doesn't depend on directly replacing return addresses and code pointers, like ROP and JOP do. Such language-specific attacks are important to consider when considering mitigations against code reuse attacks, which will be the topic of the next section.

2.4.4 Sigreturn-oriented programming

One last example of a code reuse attack that is worth mentioning here is sigreturn-oriented programming (SROP) (Bosman and Bos 2014). It is a special case of ROP where the attacker creates a fake signal handler frame and calls `sigreturn`. `sigreturn` is a system call on many UNIX-type systems which is normally called upon return from a signal handler, and restores the state of the process based

on the state that has been saved on the signal handler's stack by the kernel previously, on entry to the signal handler. The ability to fake a signal handler frame and call `sigreturn` gives an attacker a simple way to control the state of the program.

2.5 Mitigations against code reuse attacks

When discussing mitigations against code reuse attacks, it is important to keep in mind that there are two capabilities the attacker must have for such attacks to work:

- the ability to overwrite return addresses or function pointers
- knowledge of the target addresses to overwrite them with (e.g. libc function entry points).

When code reuse attacks were first described, programs used to contain absolute code pointers, and needed to be loaded at fixed addresses. The stack base was predictable, and libraries were loaded in predictable memory locations. This made code reuse attacks simple, as all of the addresses needed for a successful exploit were easy to discover.

2.5.1 ASLR

Address space layout randomization (ASLR) makes this more difficult by randomizing the positions of the memory areas containing the executable, the loaded libraries, the stack and the heap. ASLR requires code to be position-independent. Given enough entropy, the chance that an attacker would successfully guess one or more addresses in order to mount a successful attack will be greatly reduced.

Does this mean that code reuse attacks have been made redundant by ASLR? Unfortunately, this is not the case. There are various ways in which an attacker can discover the memory layout of the victim program. This is often referred to as an "info leak" (Serna 2012).

Since we can not exclude code reuse attacks solely by making addresses hard to guess, we need to also consider mitigations that prevent attackers from overwriting return addresses and other code pointers. Some of the mitigations described earlier, like stack canaries and library function hardening, can help in specific situations, but for the more general case where an attacker has obtained arbitrary read and write primitives, we need something more.

2.5.2 CFI

Control-flow integrity (CFI) is a family of mitigations that aim to preserve the intended control flow of a program. This is done by restricting the possible targets of indirect branches and returns. A scheme that protects indirect jumps and calls is referred to as forward-edge CFI, whereas a scheme that protects returns is said to implement backward-edge CFI. Ideally, a CFI scheme would not allow any control flow transfers that don't occur in a correct program execution, however different schemes have varying granularities. They often rely on function type checks or use static analysis (points-to analysis) to identify potential control

flow transfer targets. (Burow et al. 2017) compares a number of available CFI schemes based on the precision. For forward-edge CFI schemes, for example, schemes are classified based on whether or not they perform, among others, flow-sensitive analysis, context-sensitive analysis and class-hierarchy analysis.

2.5.2.1 Clang CFI

Clang's CFI includes a variety of forward-edge control-flow integrity checks. These include checking that the target of an indirect function call is an address-taken function of the correct type and checking that a C++ virtual call happens on an object of the correct dynamic type.

For example, assume we have a class `A` with a virtual function `foo` and a class `B` deriving from `A`, and that these classes are not exported to other compilation modules:

```
class A {
public:
  virtual void foo() {}
};

class B : public A {
public:
  virtual void foo() {}
};

void call_foo(A* a) {
  a->foo();
}
```

When compiling with `-fsanitize=cfi -flto -fvisibility=hidden`,[5] the code for `call_foo` would look something like this:

```
00000000004006b4 <call_foo(A*)>:
  4006b4:       a9bf7bfd        stp     x29, x30, [sp, #-16]!
  4006b8:       910003fd        mov     x29, sp
  4006bc:       f9400008        ldr     x8, [x0]
  4006c0:       90000009        adrp    x9, 400000 <_init-0x558>
  4006c4:       91216129        add     x9, x9, #0x858
  4006c8:       cb090109        sub     x9, x8, x9
  4006cc:       d1004129        sub     x9, x9, #0x10
  4006d0:       93c91529        ror     x9, x9, #5
  4006d4:       f100093f        cmp     x9, #0x2
  4006d8:       540000a2        b.cs    4006ec <call_foo(A*)+0x38>
  4006dc:       f9400108        ldr     x8, [x8]
  4006e0:       d63f0100        blr     x8
  4006e4:       a8c17bfd        ldp     x29, x30, [sp], #16
  4006e8:       d65f03c0        ret
  4006ec:       d4200020        brk     #0x1
```

[5] The LTO and visibility flags are required by Clang's CFI.

This code looks complicated, but what it does is check that the virtual table pointer (vptr) of the argument points to the vtable of `A` or of `B`, which are stored consecutively and are the only allowed possibilities. The checks generated for different types of control-flow transfers are similar.

Another implementation of forward-edge CFI is Windows Control Flow Guard, which only allows indirect calls to functions that are marked as valid indirect control flow targets.

2.5.2.2 Clang Shadow Stack

Clang also implements a backward-edge CFI scheme known as Shadow Stack. In Clang's implementation, a separate stack is used for return addresses, which means that stack-based buffer overflows cannot be used to overwrite return addresses. The address of the shadow stack is randomized and kept in a dedicated register, with care taken so that it is never leaked, which means that an arbitrary write primitive cannot be used against the shadow stack unless its location is discovered through some other means.

As an example, when compiling with `-fsanitize=shadow-call-stack -ffixed-x18`,[6] the code generated for the `main` function from the earlier stack buffer overflow example will look something like:

```
main:
    cmp w0, #2
    b.lt    .LBB1_2
    str x30, [x18], #8
    stp x29, x30, [sp, #-16]!
    mov x29, sp
    ldr x0, [x1, #8]
    bl  copy_and_print
    ldp x29, x30, [sp], #16
    ldr x30, [x18, #-8]!
.LBB1_2:
    mov w0, wzr
    ret
```

You can see that the shadow stack address is kept in `x18`. The return address is also saved on the "normal" stack for compatibility with unwinders, but it's not actually used for the function return.

2.5.2.3 Pointer Authentication

In addition to software implementations, there are a number of hardware-based CFI implementations. A hardware-based implementation has the potential to offer improved protection and performance compared to an equivalent software-only CFI scheme.

One such example is Pointer Authentication (Rutland 2017), an Armv8.3 feature, supported only in AArch64 state, that can be used to mitigate code reuse attacks.

[6] The `-ffixed-x18` flag results in treating the `x18` register as reserved, and is required by `-fsanitize=shadow-call-stack` on some platforms.

Pointer Authentication introduces instructions that generate a pointer *signature*, called a Pointer Authentication Code (PAC), based on a key and a modifier. It also introduces matching instructions to authenticate this signature. Incorrect authentication leads to an unusable pointer, that will cause a fault when used.[7] The key is not directly accessible by user space software.

Pointers are stored as 64-bit values, but they don't need all of these bits to describe the available address space, so a number of bits in the top of each pointer are unused. The unused bits must be all ones or all zeros, so we refer to them as extension bits. Pointer Authentication Codes are stored in those unused extension bits of a pointer. The exact number of PAC bits depends on the number of unused pointer bits, which varies based on the configuration of the virtual address space size.[8]

Clang and GCC both use Pointer Authentication for return address signing, when compiling with the `-mbranch-protection=pac-ret` flag. When compiling with Clang using this flag, the `main` function from the earlier stack buffer overflow example looks like:

```
main:
    cmp w0, #2
    b.lt    .LBB1_2
    paciasp
    stp x29, x30, [sp, #-16]!
    ldr x0, [x1, #8]
    mov x29, sp
    bl  copy_and_print
    ldp x29, x30, [sp], #16
    autiasp
.LBB1_2:
    mov w0, wzr
    ret
```

Notice the `paciasp` and `autiasp` instructions: `paciasp` computes a PAC for the return address in the link register (`x30`), based on the current value of the stack pointer (`sp`) and a key. This PAC is inserted in the extension bits of the pointer. We then store this signed version of the link register on the stack. Before returning, we load the signed return address from the stack, we execute `autiasp`, which verifies the PAC stored in the return address, again based on the value of the key and the value of the stack pointer (which at this point will be the same as when we signed the return address). If the PAC is correct, which will be the case in normal execution, the extension bits of the address are restored, so that the address can be used in the `ret` instruction. However, if the stored return address has been overwritten with an address with an incorrect PAC, the upper bits will be corrupted so that subsequent uses of the address (such as in the `ret` instruction) will result in a fault.

By making sure we don't store any return addresses without a PAC, we can significantly reduce the effectiveness of ROP attacks: since the secret key is not

[7] With the FPAC extension, a fault is raised at incorrect authentication.

[8] If the Top-Byte-Ignore (TBI) feature is enabled, the top byte of pointers is ignored when performing memory accesses. This restricts the number of available PAC bits.

retrievable by an attacker, an attacker cannot calculate the correct PAC for a given address and modifier, and is restricted to guessing it. The probability of success when guessing a PAC depends on the exact number of PAC bits available in a given system configuration. However, authenticated pointers are vulnerable to pointer substitution attacks, where a pointer that has been signed with a given modifier is replaced with a different pointer that has also been signed with the same modifier.

Another backward-edge CFI scheme that uses Pointer Authentication instructions is PACStack (Liljestrand et al. 2021), which chains together PACs in order to include the full context (all of the previous return addresses in the call stack) when signing a return address.

Pointer Authentication can also be used more widely, for example to implement a forward-edge CFI scheme, as is done in the arm64e ABI (McCall and Bougacha 2019). The Pointer Authentication instructions, however, are generic enough to also be useful in implementing more general memory safety measures, beyond CFI.


Add more references to relevant research

2.5.2.4 BTI

Branch Target Identification (BTI) , introduced in Armv8.5, offers coarse-grained forward-edge protection. With BTI, the locations that are targets of indirect branches have to be marked with a new instruction, `BTI`. There are four different types of BTI instructions that permit different types of indirect branches (indirect jump, indirect call, both, or none). An indirect branch to a non-BTI instruction or the wrong type of BTI instruction will raise a Branch Target Exception.


Mention more Pointer Authentication uses in later section, and add link here

Both Clang and GCC support generating BTI instructions, with the `-mbranch-protection=bti` flag, or, to enable both BTI and return address signing with Pointer Authentication, `-mbranch-protection=standard`.

Two aspects of BTI can simplify its deployment: individual pages can be marked as guarded or unguarded, with BTI checks as described above only applying to indirect branches targeting guarded pages. In addition to this, the BTI instruction has been assigned to the hint space, therefore it will be executed as a no-op in cores that do not support BTI, aiding its adoption.

2.5.2.5 CFI implementation pitfalls

When implementing CFI measures like the ones described here, it is important to be aware of known weaknesses that affect similar schemes. (Conti et al. 2015) describes how CFI implementations can suffer when certain registers are spilled on the stack, where they could be controlled by an attacker. For example, if a register that contains a function pointer that has just been validated gets spilled, the check can effectively be bypassed by overwriting the spilled pointer.

Having discussed various mitigations against code reuse attacks, it's time to turn our attention to a different type of attacks, which do not try to overwrite code pointers: attacks against non-control data, which will be the topic of the next section.

2.6 Non-control data exploits


Discuss data-oriented programming and other attacks

2.7 Hardware support for protection against memory vulnerabilities


Describe architectural features for mitigating memory vulnerabilities and for CFI

2.8 JIT compiler vulnerabilities


Write section on JIT compiler vulnerabilities

Chapter 3

Covert channels and side-channels

A large class of attacks make use of so-called side-channels. In this chapter, we focus on the the mechanisms used to make communication happen through side-channels or covert channels. In the next two chapters, we describe attacks making use of side-channels.

Side-channels and covert channels are closely related. Both side-channels and covert channels are communication channels between two entities in a system, where the entities should not be able to communicate that way.

A **covert channel** is such a channel where both entities intend to communicate through the channel. A **side-channel** is a such a channel where one end is the victim of an attack using the channel.

In other words, the difference between a covert channel and a side-channel is whether both entities intend to communicate. If one entity does not intend to communicate, but the other entity nonetheless extracts some data from the first, it is called a side-channel attack. The entity not intending to communicate is called the victim. The other entity is sometimes called the spy.

The rest of this chapter mostly describes a variety of common covert channel mechanisms. It does not aim to differentiate much on whether both ends intend to cooperate, or whether one end is a victim under attack of the other end.

3.1 Cache covert channels

Caches are used in almost every computing system. They are small and much faster memories than the main memory. They aim to automatically keep frequently used data accessed by programs, so that average memory access time improves. Various techniques exist where a covert communication can happen between processes that share a cache, without the processes having rights to read or write to the same memory locations. To understand how these techniques work, one needs to understand typical organization and operation of a cache.

3.1.1 Typical CPU cache architecture

There is a wide variety in CPU cache micro-architecture details, but the main characteristics that are important to set up a covert channel tend to be similar across most popular implementations.

Caches are small and much faster memories than the main memory that aim to keep a copy of the data at the most frequently accessed main memory addresses. The set of addresses that are used most frequently changes quickly over time as a program executes. Therefore, the addresses that are present in CPU caches also evolve quickly over time. The content of the cache may change with every executed read or write instruction.

On every read and write instruction, the cache micro-architecture looks up if the data for the requested address happens to be present in the cache. If it is, the CPU can continue executing quickly; if not, dependent operations will have to wait until the data returns from the much slower main memory. A typical access time is 3 to 5 CPU cycles for the fastest cache on a CPU versus hundreds of cycles for a main memory access. When data is present in the cache for a read or write, it is said to be a cache hit. Otherwise, it's called a cache miss.

Most systems have multiple levels of cache, each with a different trade-off between cache size and access time. Some typical characteristics might be:

- L1 (level 1) cache, 32kB in size, with an access time of 4 cycles.
- L2 cache, 256Kb in size, with an access time of 10 cycles.
- L3 cache, 16MB in size, with an access time of 40 cycles.
- Main memory, gigabytes in size, with an access time of more than 100 cycles.

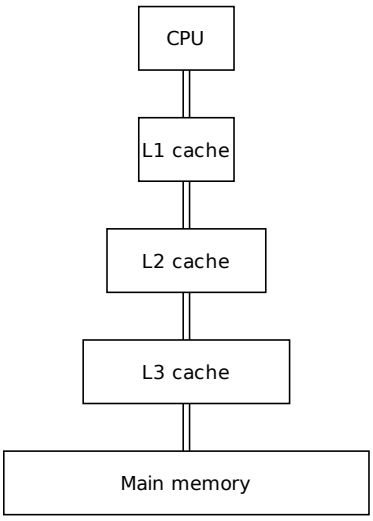


Illustration of cache levels in a typical system

If data is not already present in a cache layer, it is typically stored there after it has been fetched from a slower cache level or main memory. This is often a good decision to make as there's a high likelihood the same address will be accessed

by the program soon after. This high likelihood is known as the principle of locality.

Data is stored and transferred between cache levels in blocks of aligned memory. Such a block is called a cache block or cache line. Typical sizes are 32, 64 or 128 bytes per cache line.

When data that wasn't previously in the cache needs to be stored in the cache, most of the time, room has to be made for it by removing, or evicting, some other address/data from it. How that choice gets made is decided by the cache replacement policy. Popular replacement algorithms are Least Recently Used (LRU), Random and pseudo-LRU. As the names suggest, LRU evicts the cache line that is least recently used; random picks a random cache line; and pseudo-LRU approximates choosing the least recently used line.

If a cache line can be stored in all locations available in the cache, the cache is fully-associative. Most caches are however not fully-associative, as it's too costly to implement. Instead, most caches are set-associative. In an N-way set-associative cache, a specific line can only be stored in one of N cache locations. For example, if a line can potentially be stored in one of 2 locations, the cache is said to be 2-way set-associative. If it can be stored in one of 4 locations, it's called 4-way set-associative, and so on. When an address can only be stored in one location in the cache, it is said to be direct-mapped, rather than 1-way set-associative. Typical organizations are direct-mapped, 2-way, 4-way, 8-way, 16-way or 32-way set-associative.

The set of cache locations that a particular cache line can be stored at is called a cache set.

3.1.1.1 Indexing in a set-associative cache

For some cache covert channels, it is essential to know exactly how a memory address maps to a specific cache set.

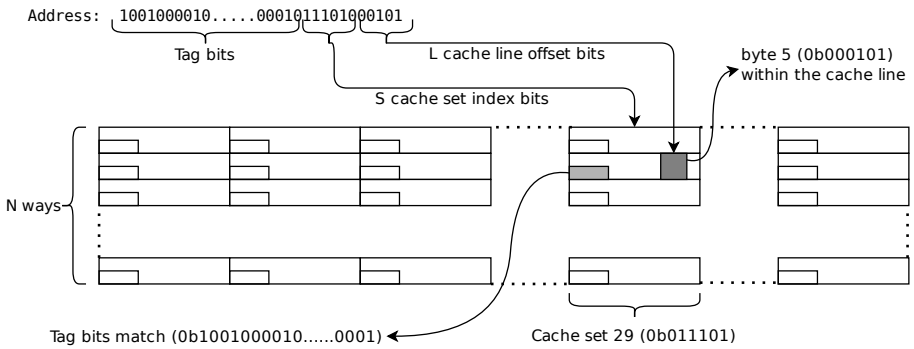


Figure 3.1: Illustration of indexing into a set-associative cache. In this example: $L = 6$ bits, hence the cache line size is $2^6 = 64$ bytes. $S = 5$ bits, so there are $2^5 = 32$ cache sets. N can be independent of address bits used to index the cache. If we assume $N = 12$ for a 12-way set-associative cache, the total cache size is $N * 2^L * 2^S = 12 * 64 * 32 = 24$KB.

Specific bits in the memory address are used for different cache indexing purposes, as illustrated in fig. 3.1. The least-significant L bits, where 2^L is the cache line size, are used to compute an address's offset within a cache line. The next S bits, where 2^S is the number of cache sets, are used to determine which cache set an address maps to. The remaining top bits are "tag bits". They are stored alongside a line in the cache so later operations can detect which specific memory address is replicated in that cache line.

For direct-mapped and fully-associative caches, the mapping of an address to cache locations also works as described above. In fully-associative caches the number of cache sets is 1, so S=0.

Also explain cache coherency ?

Also say something about TLBs and prefetching?

3.1.2 Operation of cache side-channels

Cache covert channels typically work by the spy determining whether a memory access was a cache hit or a cache miss. From that information, in specific situations, it may be able to deduce bits of data that only the victim has access to.

Let's illustrate this with describing a few well-known cache side-channels:

3.1.2.1 Flush+Reload

In a so-called Flush+Reload attack(Yarom and Falkner 2014), the spy process shares memory with the victim process. The attack works in 3 steps:

1. The Flush step: The spy flushes a specific address from the cache.
2. The spy waits for some time to give the victim time to potentially access that address, resulting in bringing it back into the cache.
3. The Reload step: The spy accesses the address and measures the access time. A short access time means the address is in the cache; a long access time means it's not in the cache. In other words, a short access time means that in step 2 the victim accessed the address; a long access time means it did not access the address.

Knowing if a victim accessed a specific address can leak sensitive information. Such as when accessing a specific array element depends on whether a specific bit is set in secret data. For example, (Yarom and Falkner 2014) demonstrates that a Flush+Reload attack can be used to leak GnuPG private keys.

Should there be a more elaborate example with code that demonstrates in more detail how a flush+reload attack works?

3.1.2.2 Prime+Probe

In a Prime+Probe attack, there is no need for memory to be shared between victim and spy. The attack works in 3 steps:

1. The Prime step: The spy fills one or more cache sets with its data, for example, by accessing data that maps to those cache sets.
2. The spy waits for some time to let the victim potentially access data that maps to those same cache sets.
3. The Probe step: The spy accesses that same data as in the prime step. Measuring the time it takes to load the data, it can derive how many cache lines the victim evicted from each cache set in step 2, and from that derive information about addresses that the victim accessed.

(Osvik, Shamir, and Tromer 2005) which first documented this technique in 2005 demonstrates extracting AES keys in just a few milliseconds using Prime+Probe.

3.1.2.3 General schema for cache covert channels

An attentative reader may have noticed that the concrete named attacks above follow a similar 3-step pattern. Indeed, (Weber et al. 2021) describes this general pattern and uses it to automatically discover more side-channels that follow this 3-step pattern. They describe the general pattern as being:

1. An instruction sequence that resets the inner CPU state (*reset sequence*).
2. An instruction sequence that triggers a state change (*trigger sequence*).
3. An instruction sequence that leaks the inner state (*measurement sequence*).

Other cache-based side channel attacks following this general 3-step approach include: Flush+Flush(Gruss, Maurice, Wagner, et al. 2016), Flush+Prefetch(Gruss, Maurice, Fogh, et al. 2016), Evict+Reload(Percival 2005), Evict+Time(Osvik, Shamir, and Tromer 2005), Reload+Refresh(Briongos et al. 2020), Collide+Probe(Lipp et al. 2020), etc.

3.2 Timing covert channels

3.3 Resource contention channels

3.4 Channels making use of aliasing in branch predictors and other predictors

Should we also discuss more "covert" channels here such as power analysis, etc?

Chapter 4

Physical access side-channel attacks

Write chapter on physical access side-channel attacks.

Chapter 5

Remote access side-channel attacks

This chapter covers side-channel attacks for which the attacker does not need physical access to the hardware.

5.1 Timing attacks

An implementation of a cryptographic algorithm can leak information about the data it processes if its run time is influenced by the value of the processed data. Attacks making use of this are called timing attacks.

The main mitigation against such attacks consists of carefully implementing the algorithm such that the execution time remains independent of the processed data. This can be done by making sure that both:

a) The control flow, i.e. the trace of instructions executed, does not change depending on the processed data. This guarantees that every time the algorithm runs, exactly the same sequence of instructions is executed, independent of the processed data.

b) The instructions used to implement the algorithm are from the subset of instructions for which the execution time is known to not depend on the data values it processes.

   For example, in the Arm architecture, the Armv8.4-A DIT extension guarantees that execution time is data-independent for a subset of the AArch64 instructions.

   By ensuring that the extension is enabled and only instructions in the subset are used, data-independent execution time is guaranteed.

At the moment, we do not know of a compiler implementation that actively helps to guarantee both (a) and (b). A great reference giving practical advice on how to achieve (a), (b) and more security hardening properties specific for cryptographic kernels is found in (Pornin 2018).

As discussed in (Pornin 2018), when implementing cryptographic algorithms, you also need to keep cache side-channel attacks in mind, which are discussed in the section on cache side-channel attacks.

5.2 Cache side-channel attacks

Write section on cache side-channel attacks. See the first comment on PR24 for suggestions of what this should contain.

Chapter 6

Supply chain attacks

A software *supply chain attack* occurs when an attacker interferes with the software development or distribution processes with the intention to impact users of that software.

Supply chain attacks and their possible mitigations are not specific to compilers. However, compilers are an attractive target for attack because they are widely deployed to developers, in continuous integration systems and as JITs. Also, an infected compiler has the possibility to make a much larger impact if it can silently spread the infection to other software created with or run using it.

This chapter explores the history of supply chain attacks that involve compilers and what can be done to prevent them.

6.1 History of supply chain attacks

As far back as 1974 Karger & Schell theorized about an attack on the Multics operating system via the PL/I compiler (Paul A. and Roger R. 1974). In this attack, a trap door is inserted into the compiler, which then injects malicious code into generated object code. Furthermore, the trap door could be designed to reinsert itself into the compiler binary so that future compilers are silently infected without needing changes to their source code. This attack method was subsequently popularised by Ken Thompson in his 1984 ACM Turing Award acceptance speech *Reflections on Trusting Trust* (Thompson 1984).

If these cases seem far-fetched then consider that there have been several real examples of supply chain attacks on development tools.

Induc is a family of viruses that infects a pre-compiled library in the Delphi toolchain with malicious code (Gostev 2009). When Delphi compiles a project the malicious library is included into the resulting executable, thus enabling the virus to spread. The virus was first detected in 2009 and was circulating undetected for at least a year beforehand. Several popular applications are known to have been infected, including a chat client and a media player. Overall, in excess of a hundred thousand infected computers were detected world-wide by anti-virus solutions.

XcodeGhost is the name given to malware first detected in 2015 that infected thousands of iOS applications (Cox 2015). The source of the infection was tracked down to a trojanized version of Xcode tools. The malware exists in an extra object file within the Xcode tools and is silently linked into each application as it is built. File sharing sites were used to spread the trojanized Xcode tools to unwitting developers.

A trojanized linker was found to be involved in a supply chain attack discovered in 2017 named ShadowPad (Greenberg 2019). Some instances of the attack were perpetrated using a trojanized Visual Studio linker that silently incorporates a malicious library into applications as they are built. Related attacks named CCleaner and ShadowHammer used the same approach of a trojanized linker to infect built applications. Infected applications from these attacks were distributed to millions of users world-wide.

These cases highlight that attacks on compilers, and especially linkers and libraries, are a viable route to silently infect many other applications, and there is no doubt that there will be more such attacks in the future. Let us now explore what we can do about these.

Explain how these vulnerabilities arise and how to mitigate them.

Chapter 7

Other security topics relevant for compiler developers

Write chapter with other security topics.

Write section on securely clearing memory in C/C++ and undefined behaviour.

Appendix: contribution guidelines

Write chapter on contribution guidelines. These should include at least: project locaton on github; how to create pull requests/issues. Where do we discuss - mailing list? Grammar and writing style guidelines. How to use todos and index.

Index

Todo list

References

Aleph One. 1996. "Smashing the Stack for Fun and Profit." 1996. http://www.phrack.org/issues/49/14.html#article.

Beer, Ian. 2020. "An iOS Zero-Click Radio Proximity Exploit Odyssey." 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

Beer, Ian, and Samuel Groß. 2021. "A Deep Dive into an Nso Zero-Click iMessage Exploit: Remote Code Execution." 2021. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html.

Bletsch, Tyler, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. "Jump-Oriented Programming: A New Class of Code-Reuse Attack." In *Proceedings of the 6th Acm Symposium on Information, Computer and Communications Security*, 30–40. ASIACCS '11. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1966913.1966919.

Bosman, Erik, and Herbert Bos. 2014. "Framing Signals - a Return to Portable Shellcode." In *2014 Ieee Symposium on Security and Privacy*, 243–58. https://doi.org/10.1109/SP.2014.23.

Briongos, Samira, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. 2020. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks." In *29th Usenix Security Symposium (Usenix Security 20)*. https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

Burow, Nathan, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. "Control-Flow Integrity: Precision, Security, and Performance." *ACM Comput. Surv.* 50 (1). https://doi.org/10.1145/3054924.

Conti, Mauro, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. "Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks." In *Proceedings of the 22nd Acm Sigsac Conference on Computer and Communications Security*, 952–63. CCS '15. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2810103.2813671.

Cox, Joseph. 2015. "Hack Brief: Malware Sneaks into the Chinese iOS App Store." *WIRED*. https://www.wired.com/2015/09/hack-brief-malware-sneaks-chinese-ios-app-store/.

Dowd, Mark, John McDonald, and Justin Schuh. 2006. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional.

Dullien, Thomas. 2020. "Weird Machines, Exploitability, and Provable Unexploitability." *IEEE Transactions on Emerging Topics in Computing* 8 (2): 391–403. https://doi.org/10.1109/TETC.2017.2785300.

Gostev, Alexander. 2009. "A Short History of Induc." 2009. https://securelist.com/a-short-history-of-induc/30555/.

Greenberg, Andy. 2019. "Supply Chain Hackers Snuck Malware into Videogames." *WIRED*. https://www.wired.com/story/supply-chain-hackers-videogames-asus-ccleaner/.

Groß, Samuel. 2020. "JITSploitation I: A Jit Bug." 2020. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html.

Gruss, Daniel, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. "Prefetch Side-Channel Attacks: Bypassing Smap and Kernel Aslr." In *Proceedings of the 2016 Acm Sigsac Conference on Computer and Communications Security*. CCS '16. https://doi.org/10.1145/2976749.2978356.

Gruss, Daniel, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. "Flush+Flush: A Fast and Stealthy Cache Attack." In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, 279–99. DIMVA 2016. https://doi.org/10.1007/978-3-319-40667-1_14.

Hicks, Michael. 2014. "What Is Memory Safety?" 2014. http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

Hu, Hong, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks." In *2016 Ieee Symposium on Security and Privacy (Sp)*, 969–86. https://doi.org/10.1109/SP.2016.62.

Liljestrand, Hans, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. 2021. "{PACStack}: An Authenticated Call Stack." In *30th Usenix Security Symposium (Usenix Security 21)*, 357–74.

Lipp, Moritz, Vedad Hadzic, Michael Schwarz, Arthur Perais, Clementine Lucie Noemie Maurice, and Daniel Gruß. 2020. "Take a Way: Exploring the Security Implications of Amd's Cache Way Predictors." In *Proceedings of the 15th Acm Asia Conference on Computer and Communications Security, Asia Ccs 2020*. https://doi.org/10.1145/3320269.3384746.

McCall, John, and Ahmed Bougacha. 2019. "Arm64e: An Abi for Pointer Authentication." 2019. https://llvm.org/devmtg/2019-10/slides/McCall-Bougacha-arm64e.pdf.

Miller, Matt. n.d. "Modeling the Exploitation and Mitigation of Memory Safety Vulnerabilities." Breakpoint 2012. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2012_10_Breakpoint/BreakPoint2012_Miller_Modeling_the_exploitation_and_mitigation_of_memory_safety_vulnerabilities.pdf.

Osvik, Dag Arne, Adi Shamir, and Eran Tromer. 2005. "Cache Attacks and Countermeasures: The Case of Aes." In *IACR Cryptology ePrint Archive*. https://doi.org/10.1007/11605805_1.

Paul A., Karger, and Schell Roger R. 1974. "MULTICS Security Evaluation: VULNERABILITY Analysis," 52. https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/karg74.pdf.

Percival, Colin. 2005. "Cache Missing for Fun and Profit." BSDCan. https://eprint.iacr.org/2005/271.

Pornin, Thomas. 2018. "Why Constant-Time Crypto?" 2018. https://www.bearssl.org/constanttime.html.

Rutland, Mark. 2017. "ARMv8.3 Pointer Authentication." 2017. https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf.

Schuster, F., T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. 2015. "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications." In *2015 Ieee Symposium on Security and Privacy (Sp)*, 745–62. Los Alamitos, CA, USA: IEEE Computer Society. https://doi.org/10.1109/SP.2015.51.

Seacord, Robert C. 2013. *Secure Coding in c and C++*. 2nd ed. Addison-Wesley Professional.

Serna, Fermin J. 2012. "The Info Leak Era on Software Exploitation." 2012. https://www.youtube.com/watch?v=VgWoPa8Whmc.

Shacham, Hovav. 2007. "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*, 552–61. CCS '07. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1315245.1315313.

Sharma, Siddharth. 2014. 2014. https://www.redhat.com/en/blog/enhance-application-security-fortifysource.

Solar Designer. 1997. "Getting Around Non-Executable Stack (and Fix)." 1997. https://seclists.org/bugtraq/1997/Aug/63.

Thompson, Ken. 1984. "Reflections on Trusting Trust." https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf.

Weber, Daniel, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. "Osiris: Automated Discovery of Microarchitectural Side Channels." In *USENIX Security'21*. https://arxiv.org/abs/2106.03470.

Yarom, Yuval, and Katrina Falkner. 2014. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In *23rd Usenix Security Symposium (Usenix Security 14)*, 719–32. San Diego, CA: USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.