Systems and Software Security

Josh Felmeden

November 11, 2021

Contents

1 Overview

We learn about this topic so that we can avoid our own software having these same exploits.

So, what is a program?

- Functional (intended) behaviour
- Security policy (what it's not meant to do)

Unintended behaviours can include:

- Design flaws
- Bugs
- Lower-level bugs
- Mistaken assumptions

1.1 Weaknesses and Vulnerabilities

A **weakness** is when a program has a flaw that allows an attacker to do something the programmer didn't anticipate, or which could cause problems.

A **vulnerability** is when these weaknesses can be *exploited* by an attacker to violate part of the program's design and do something harmful.

Weakness is *not* a vulnerability

Just because a program has a weakness does not mean it is exploitable.

An **exploit** is a program or technique that takes advantage of a vulnerability to violate the security policy. They can be published to prove existence of a vulnerability or utilised as part of malware.

- High-level code gets translated into a low-level representation
- Separate variables become continuous memory addresses
- Data types become bit-patterns
- Memory corruption becomes a big problem

And typical vulnerabilities we will see are:

- Over/underflow
- Data corruption
- Control flow corruption
- Denial of service

These normally cause the program to crash, but occasionally they can become an *exploit* where we can gain access to places we shouldn't have.

1.2 Mitigations

We can put in place mechanisms that remedy the weakness, or prevent the exploitation of the vulnerability. For example, stack canaries let us spot when a stack buffer has overflown. Note that it doesn't fix the buffer overflow but it makes it a **lot harder** to exploit. We can also randomise where memory is kept (ASLR), shadow stacks, sandboxing (such as a firewall).

1.3 The C programming language

We will mostly be looking at C in this module because it's a really popular programming language. It's not dead, honest!!! Also, pretty much everything is built on top of it.

It's designed for systems programming and is unsafe *by design*. It is therefore the programmers job to ensure that their program is correct, allowing the programmer to access raw(ish) memory addresses (pointers).

People don't like C (me included) because it always assumes the programmer knows best. It has limited support for anything more than primitive types, and even some primitive types have limited support. It also has limited bounds settings and setting a variable to `const` doesn't actually make it a constant, because you can still edit the variable if you know where it's stored in memory.

It's not all bad, though. A lot of legacy code is still written in C. Some effort has been made to rewrite this code in safer languages, but this isn't always possible or even a good idea. While C is very stable and portable and really useful, it can lead to bugs (though not all bugs relate to Cs unsafeness, some of it could be the programmer being a dummy). Rewriting C could lead to whole new bugs and oversights.

1.4 Assembly

1.4.1 Memory Layout

While we can generalise, it is important to note that not all memory looks the same. Different architectures and OSs might have memory look different.

From low to high:

- .text (Program code)
- .plt (Library code)
- .data (initialised data)
- .bss (uninitialised Data)
- The heap (growing up)

From high to low:

- Arguments and environment

- The stack (growing down)

NOTE: Stack goes down, heap goes up.

1.4.2 x86 Assembly (32-bit)

There are 6 32-bit general purpose general registers: `eax, ebx, ecx, edx, esi, edi`, 2 special 32-bit registers: `esp, ebp` and 1 instruction pointer: `eip`. There are sometimes more registers depending on the chip and also tonnes of instructions, since there's a pretty big CISC (this normally gets translated into a RISC microcode, but not always)

1.4.3 amd64 Assembly

There are 16 64-bit general purpose registers: `raz, rbx, rcx, rdx, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15`, 2 special 32-bit registers: `rsp, rbp` and 1 instruction pointer `rip`. Again, there can sometimes be more registers depending on the chip and heaps of instructions (which NORMALLY get translated into RISC but sometimes doesn't. Look at the manual if you want to know CHRIST).

1.4.4 x86/64 Assembly

There are lots of different assemblers for x86, each with their own syntax. There are strong opinions about what is better, but you need to kind of get a feel for what works for you.

1.5 Calling Conventions

Calling conventions handle how functions are called from C, translation of this into registers, where arguments go for shared libraries, etc.

It's defined by the OS but not strictly enforced. Most programming languages follow the rules set by C.

There are a lot of different x86 calling conventions, and you pretty much just have to look up whatever your system uses (Windows uses more than one, helpfully).

In essence:

- `cdecl`: everything goes on the stack, caller cleans up
- `stdcall`: everything goes on the stack, callee cleans up
- `fastcall`: pass things in registers `eax, edx, ecx` then on the stack
- `thiscall`: class pointer in `exc` then stack (usually for c++ or Windows)

1.5.1 amd64 Calling conventions

With amd64, the instruction set designers sorted a lot of the mess out and started again. Now, we only have two (kind of three) conventions, similar to fastcall. Again, look it up.

1.6 Useful Tools

- Debuggers: **GDB** or LLDB
- Disassemblers: Ghidra, Radare2, Objdump
- Languages: Python
- Hex Editors: Radare2, XXD, emacs???/vi

Compilation Options

- For GCC
  - `-fno-stack-protector`
  - `-z execstack` (run shellcode off the stack)
  - `-mno-accumulate-outgoing-args` (don't optimise calling conventions)

2 Software Vulnerabilities and Attacks Part 1

2.1 Buffer Overflows

When you declare an array in C, you get a region of memory. Pointers are used to address arrays, and it's very easy to fall off the edge of this region. They have been known about since the dawn of computers, so it's nothing new.

To understand buffer overflows, we need to understand how functions work. We write from the top of the stack to the bottom of the stack. So, when we go into a function, we push the memory address of the stack before the function call onto the stack. Once we've done that, we push the variables of the function onto the stack. This is the basic idea for memory layout for stacks.

Now, what if it got a little more interesting?:

```c
//example 1.c
void function(char *str) {
    char buffer[16];

    strcpy(buffer,str)
}

void main() {
    char large_string[256];
    int i;

    for(i = 0; i < 255; i++) {
        large_string[i] = 'A';
    }
```

```
    function(large_string);
}
```

The memory layout would look like this:

```
TOS                                                 BOS
           buffer          sfp     ret  *str
<-------    [AAAAAAAAAAAAA][AAAAA][AAAAAA][AAAAAAAAAAAAAAA...]
```

Since `strcpy` only deals with pointers, we just start writing 'A' into the buffer, and once it reaches the end of the buffer, it just keeps writing. Now, once the function is finished, it returns. When it attempts to read the memory address for the return, it's going to try to return to 'AAAAAA', which will probably crash the function.

Being able to overwrite stack data is bad, but overwriting return addresses gives us arbitrary code execution. Normally, it just causes an access validation, or a bad instruction. But, sometimes, you can take over the program.

2.1.1 shellcode

The classic way of doing this is with buffer shellcode. This rarely works now, but you can turn off the protectors that stop this happening. THe modern way of doing this is *return oriented programming (ROP)* and we'll visit this later.

There are some tricks to make it easier:

- Alphabetic shellcode
- NOP-sleds (instructions that do nothing, padding the addresses)

2.1.2 Prevention methods

- Stack canaries spot if buffers have been overrun.
- WX̂ (write xor execute) makes shellcode harder (but not impossible)
- Use bounded data structures and not the old C ones
- Use the bounded memory functions (`strncpy`)
- Use a modern compiler toolchain and turn on all the security features

2.2 Format Strings

A format string is a vulnerability in C-style print functions. It allows an attacker to read from the stack and other places. It also allows an attacker to write to any memory addresses on the stack.

With `printf`, if we don't put enough arguments, such as `printf("Hello %s!  \n)`, we would get a warning, because the compiler can't be sure that it is wrong.

If we then combine this with something like `gets`, we are able to access the stack arbitrarily, and even write to it with %n

To fix this, we can just listen to the compiler warnings. Some modern systems remove the functionality with it, while others log its use.

2.3 Race Conditions

Computers can do more than one thing at once, and sometimes the order gets messed up which can lead to bugs. Here's a really simple increment function:

```
void increment(int *n) {
    int temp;

    temp = *n;
    temp += 1;
    *n = temp;
}
```

This isn't thread safe, however, because if we are not careful we can lose increments. If two users call this really quickly, we might lose one of the increments. This, at the moment, is only a correctness issue. How does it become a security issue?

The *access* system call checks the accessibility of the file named by the path argument for the access permissions indicated by the mode argument. If we have a suid-program that does controlled writes as root, then it checks using access if your real user can write to a file, then does the writing as root. To avoid this kind of race condition, we can just use synchronisation around time-of-check and time-of-use. These kinds of bugs are really quite dangerous and hard to deal with, though.

3 OS Security

3.1 What is an OS?

An operating system provides an abstraction over the computer's hardware. Bigger OSs have to run more than one program with more than one user. We normally like it to implement some security policies.

Access Control Security Goals are essentially:

- **Confidentiality**: you can't see what you don't need to see
- **Integrity**: you can't tamper with stuff that's not yours
- **Availability**: you can get at your stuff.

These goals are interdependent: if I can tamper with data, who cares if it is confidential?

A **principal** is a person describing the access control policy or human trying to follow the policy.

Object is the resources that we are writing the policy about.

Subjects are the things (processes) interacting with the objects that we are trying to restrict.

3.1.1 UNIX DAC — Discrentionary Access Controls

This is the traditional access control mechanism present in almost all OSs in some form. Objects have an owner and a group. At the owner's *discretion*, they can say what they, the group, and everyone else can do with the object (read, write, execute).

There are some flaws with DAC, unfortunately. Imagine a user (Alice) wants to run a web browser. We would like that to be able to access her downloads folder, but probably not the SSH keys.

Now, imagine Alice wants to run an SSH server. We want her to be able to access her SSH keys but probably not the downloads folder.

In essence, the DAC policy is described at the object level. We could work around it, so Alice's programs run as an Alice-unprivileged user and use the group permissions to set where they can access, and then duplicate the policy for multiple users, so this isn't really viable since it gets so complicated really fast, as well as being hard to verify. This doesn't mean it's impossible, and some systems do utilise this.

The other problem is that do we trust the sysadmin to get the policy right? We need a mechanism to be able to enforce a security policy from the top down, and not just rely on discretionary controls. This is the *principle of least privilege*.

3.1.2 Reference Monitors

These reference monitors are going to help us fix this dilemma. We will still have *subjects*, but processes are associated with a security context (user, group, privileges). We will also still have *objects*, and these will have security information (DAC and xattrs (extra attributes)).

On login, processes get the capabilities of their principal, and then these are progressively dropped. Processes also inherit the capabilities of the process that made them.

There is no way for subjects to access objects except through the reference monitor (complete mediation). When a subject makes a system call:

- Get information about the subject
- Get information about the object
- Apply the system policy based on the information
- Log that a decision was made
- Return the decision

Race conditions can crop up in this, so be aware of that.

3.1.3 MAC — Mandatory Access Controls

The sysadmin sets the access control policy (which may just be the DAC). The simplest form in *multi level security*, which emerged from the US military. Subjects and objects associated with a security level:

- Unclassified
- Confidential
- Secret
- Top secret

This is the usual hierarchy of levels, but might have more or less.

One security model is the **Bell-LaPadula** method, meaning people can't read above their security clearance, or write to a security level lower than their level (don't want someone accidentally leaking data to lower levels). This method focuses on *confidentiality*.

Another model is the **Biba** method. This is a no read-down, no write-up. This preserves *integrity*.

3.2 Linux Security Modules

The solution to Linux's security is the Linux Security Model (LSM) framework. This implements a reference monitor for Linux. It has dynamically loadable kernel module hooks into system call checks. The framework is verified, and modules are (in theory) small and verifiable. The hook function returns access descision:

- `0`: Access granted
- `ENOMEM`: No memory available
- `EPERM`: Not enough privileges.

3.2.1 SELinux

One we are going to look at in detail is **SELinux**. This is a framework originally developed by the NSA. It's based around type-based enforcement and RBAC (role based access controls).

The types of hooks possible are:

- Management hooks
  - Called to handle object lifecycle
- Path-based hooks
  - Related to pathnames
- Object-based hooks
  - Path kernel structure corresponding to objects

Need a mechanism to interact with SELinux from userland. This enables the filesystem to load policies and configuration. It also gets audit data.

All subjects get labeled with a security context:

- User
- Domain
- Role

Rules describe what each *subject* domain can do with an *object* domain. They can get a bit complicated. An example of this is `/etc/passwd`, where the user information readable by any user. Or `/etc/shadow` password information readable by root only. The way this is done looks like this:

```
'normal users are allowed to read normal files
allo user_t public_t : file read

'users in the password_t domain can r/w files in the password_data_t domain
allow passwd_t passwd_data_t : file {read write}

'allow users to actually run the password program, and transition their domain
allow user_t passwd_exec_t : file execute
allow user_t passwd_t : process transition
type transition user_t passwd _exec_t : process passwd_t
```

This seems very complicated, but it kind of makes sense. The rule design is very hard.

3.3 Intrusion detection

Intrusion detection is a service that monitors a system and looks for unusual or failed attempts to access system resources:

- Could be a single event, could be a combination
- Could be probabilistic
- Could be running on the host
- Could be running on the network.
- Usually attempting to do detection in real-time (or near)

We are looking for failed authentication attempts or odd network traffic. Another thing we are looking for is users running unusual processes, or accessing unusual files. Essentially, anything unusual should be flagged.

Here are some types of intrusion detection systems

- Host based
  - Runs as a privileged process on the host
  - Uses information from the OS/reference monitor
- Network based
  - Runs either on the host or on the networ
  - Looks at network traffic, who is contacting who and how often
- Signature based
  - Identify attacks based on known attack patterns
- Anomaly based

  - Identify attacks based on a machine-learning model of what is normal for a given user or process

False positives or negatives are really annoying but they can be fed into the rules for next time.

The goals of the IDS are to run continuously and resist attempts to subvert mechanisms. It shouldn't make the system unusable in terms of performance or usability overhead. It should adapt to changes in a system's use and reconfiguration. It should also scale to work with big systems. It should degrade gracefully and not fail (ideally).

4 Software Vulnerabilities and Attacks 2

4.1 Heap overflow

Heap based overflows involve a discussion of `malloc`, unfortunately. This is *very* system dependent and has changed a lot over time. Therefore, we are going to go *high-level* and describe the concepts and history. To understand in detail, the implementation of the system must be researched.

While we normally allocate memory with `malloc` and `free`, they are actually using something called `mmap` (memory map). This command works via the kernel to assign and manage regions of memory. But, system calls are expensive and creating or new-ing objects dynamically is really common. C is supposed to be pretty portable, and since not all OSs implement POSIX APIs portably, we instead manage memory via the user, as opposed to the kernel.

When a program starts, we give it a large region of memory somewhere in its virtual address space and an API for managing it. It can call the lower-level system calls if necessary. Data structures to manage things were initially based on a heap, so let's call this the heap and we keep it as far away as possible to avoid things bumping into each other.

So, `malloc` and `free` are the libC API for dynamically assigning memory for objects. The essential idea is:

- Ask for memory with `malloc/calloc`
- Mark it as used with free
- Dynamically grow or change with `realloc`

Heap overflows are kind of not realistic, so we will look at Glibc `malloc`.

4.1.1 Glibc Malloc

Memory starts out as a big empty array (called an arena). When malloc is called, put the following chunk data structure on the heap. Return the pointer to the start of the payload. Data gets more and more chunked as time goes on. On free, write some data into the old payload, including a pointer to the next chunk forward and a pointer to the last point back. There are various sizes, but sequential. When freeing memory, check the forward and back pointer, if the previous chunk is also freed, then

merge the two chunks together and update the length to be combined (headers).

We can attack this via making a chunk that looks like it's already been freed. We can set headers in our own tables, since we know that the size field will be added to the address before a pointer you control. If we manage to do an arbitrary write (return address) we can completely compromise the system. This is a lot of work for a single integer write, but sometimes this is all you need.

4.2 Return oriented Programming

We looked at *smashing the stack* earlier, as well as *injecting shellcode*. But, this doesn't really work anymore, since OSs mark memory sections as marked, so injecting shellcode is a thing of the past (since the 90s really). But, why do we need to write a program into memory at all?

Shellcode itself simply sets up registers, pushes the location of the shell to the stack, gets the stack pointer, and then calls `execve`. There is already a command for doing this in C, however, called `system()`. This function:

- Runs a program in the system shell
- so there *must* be a `/bin/sh` string already in memory to pass to the exec syscall
- If we know its address, do we even need to push it on?

The basic idea is that instead of injecting the shellcode, we can set up the stack so that it looks like the arguments to a call to `system()` and assume the cdecl calling convention. Therefore, instead of returning onto our shellcode, we'll return into the libc `system()` function. To do this, we need a few things:

- System needs to be already loaded into memory
- ASLR can be problematic, but depends on how its implemented

Unfortunately for the attacker, the fixes for this are pretty easy. AMD64 architecture doesn't pass arguments on the stack by default. If more randomisation is added, it's harder to guess library functions. Also, ASCII armour strings are in memory by XORing them with patterns to make them harder to steal.

Remember turing machines from second year? It turns out that if you make it in a certain way, you can make universal computation. So, wouldn't it be really unfortunate if you could make a Turing machine out of the instructions right before the return instruction in a program's memory?

Turns out, you can do this, and this is called **Return Oriented Programming (rop)**. We know a buffer overflow gives us control over the stack, instead of overwriting just *one* return address, we can write a series of stack frames. Each saved instruction pointer will be to an instruction just before a return instruction. This is called **gadgets**. Instead of writing shellcode, we find a series of gadgets that when run in sequence, have the same effect. If we manage to find a set of gadgets that is Turing complete, we can reuse the existing program code to implement ANY shellcode without injecting the actual shellcode.

Right then, whats the plan of attack?

1. Find a gadget for '`pop rdi' ret`'
N-1. Setup stack as `&(pop rdi' ret) | &("/bin/sh") | &system`
N. Return

We also need to defeat ASLR, since libraries usually get dynamically loaded into memory by mmaping the whole library into memory. If we can leak a pointer of something within the pointers where all the functions are in a library (held in the .got file), we can learn where the pointers are. So, new plan of attack:

1. Find a gadget for '`pop rdi; ret`'
2. Find .got entry for the puts function
3. Leak it
4. Recall main (so we don't randomise memory)
5. Calculate libc's ASLR offset and where memory addresses really are
6. Setup stack as `&(pop rdi' ret) | &("/bin/sh") | &system`
7. Return

5 Software Defence

5.1 Program Analysis

We need to ensure that the software development lifecycle is secure. It's not easy to find bugs manually in large scale programs.

Manual testing can only go so far. If we have loops, it can become quite difficult to see all of the possible execution paths, even in very small programs. Static analysis, therefore, is very difficult.

Program analysis is the automatic process of analysing the behaviour of computer programs regarding a property such as correctness, reliability, safety, and security. The types we will see is:

- Static Analysis: performed without executing the program
- Dynamic Analysis: performed at runtime
- Hybrid: a mix of the two

5.1.1 Static Program Analysis

This method of analysis is just looking at the code without executing it. This becomes really quite difficult with large scale programs. For things like memory allocation, you might not even know how that will work (random analysis etc.). Binary code is even more challenging because actually understanding it would take such a long time. Compilers make heavy use of this analysis to ensure correctness of programs.

This is beneficial because we can analyse every component and path of the application if we have

access to the code.

The tools we have access to are:

- LLVM
- For binary code we have a few: IDA, Ghidra, Miasm, angr...

5.1.2 Dynamic Progam Analysis

This method of analysis is essentially like debugging — it is analysed at runtime. Dynamic analysis is both very precise and scales very well. However, it is limited to the executed code of the program, so coverage is a problem.

The benefits are that we can look at things like the dynamic allocation of memory and profiling and so on.

The tools we have access to are:

- Intel pin, Dyninst, Valgrind
- Security tools (...)

5.1.3 Soundness and Completeness

We have two separate conditions we need to check for when analysing a program: **soundness** and **correctness**.

- *Soundness* is essentially saying that if analysis says no bugs, there are no bugs and vice versa. If analysis says true $\rightarrow$ true.
- *Completeness* essentially states that if there are no bugs, analysis will say there are no bugs. True $\rightarrow$ analysis says True.

	Complete	**Incomplete**
Sound	• Reports all errors • Reports no false alarms **Undecidable**	• Reports all errors • May report false alarms **Decidable**
Unsound	• May not report all errors • Reports no false alarms **Decidable**	• May not report all errors • May report false alarms **Decidable**

In most cases, the program will be both unsound and incomplete.

5.1.4 Generic Approach

We need to decide whether our program is intraprocedural or interprocedural:

- Intra — per function analysis (ignoring side effect of function calls)
- Inter — Function analysis (considering side effects of function calls)

Next, we generate a control flow graph (CFG), and then optionally generate the interprocedural CFG (ICFG) and finally we data-flow analyse the generated CFGs.

5.1.5 Intermediate Representation

We can represent each complex statement that we have via 'high-level' assembly code. This will contain:

- Binary logic and arithmetic operators
- Use of temporary memory locations
- Assignment to variables, temporary locations
- A label assigned to each instruction

E.g.:

```
var1 = (var2 + var3)  + func(A)
' Translates to
L1: t1 = var2 + var3
L2: t2 = func(A)
L3: var1 = t1+t2
```

As shown, it is called 3-address code because we only use 3 memory addresses. This is maintained by creating new temporary variables (in this example, `t1` and `t2`).

5.1.6 Basic Block (BB)

A maximal sequence of instructions with single entry and exit. Execution of BB is *atomic* under normal conditions.

5.1.7 Control Flow Graph

Control flow graphs are a representation of how the execution may progress inside a given function It is a graph (V, E) such that:

- $V = \{B_i\}$ where B_i is a basic block.
- $E = \{B_i, B_j\}$ where the last instance of B_i is a jump to the first instance of B_j and the first instance of B_j follows the last instance of B_i in the TAC.

5.1.8 Call Graph

A call graph is computed for the whole program and is represented as a directed graph (V, E) such that:

- $V = \{F_i\}$ where F_i is a function
- $E = \{(F_i, F_j)\}$ where F_i calls F_j.

5.1.9 Redundant Expressions

An expression is redundant at a location if:

- It is computed at location i
- This expression is computed on every path going from initial location to location i
- On each of these paths, operands of e are not modified between the last computation of e and location i.

Optimisation is performed as follows:

- Computation of the available expressions (via data flow analysis)
- $x := e$ is redundant at location i if e is available at i
- $x := e$ is replaced by $x := t$ (where t is a temp memory address containing the value of e).

We then look at the variables to see if they are changing to evaluate whether they are redundant or not.

5.1.10 Dataflow Equation for Available Expressions

For a basic block b we note:

- $In(b)$: available expressions when entering b
- $Kill(b)$: expressions made *non-available* by b (because an operand of e is modified by b)
- $Gen(b)$: expressions made *available* by b (computed in b and operands not modified afterwards)
- $Out(b)$ available expressions when exiting b

$$Out(b) = (In(b) \backslash Kill(b)) \cup Gen(b) = F_b(In(b))$$

Where F_b is a **transfer function** of block b.

To compute $In(b)$:

- If b is the initial block:

$$In(b) = \emptyset$$

- If b is not the initial block, an expression e is available at its entry point iff it is available at the exit point of *each* predecessor of b in the CFG

$$In(b) = \bigcap_{b' \in Pre(b)} Out(b')$$

This is called forward data-flow analysis along the CFG paths.

5.1.11 Reaching Definition

Every assignment is a definition. A definition d reaches a point p if there exists a path from the point immediately following d to p such that d is not killed (overwritten) along that path.

5.2 Dynamic Analysis

We have already looked at the static version of analysis, and this is useful, but it leaves some gaps in the space that we have not analysed. As previously discussed, there may be some functions that are never statically referenced in previously visited code. The solution to this is to run dynamic analysis. The program should be run multiple times and observe the targets of indirect code jumps and calls.

Dynamic analysis is a technique that is performed at runtime and has historically been used for performance monitoring and software testing. Security related behavioural analysis is all based on dynamic analysis.

We can monitor call instructions at runtime using **GDB**. We can set a breakpoint at each function call before the program starts and then look at the stack. But, this is a lot of effort. Instead, we can use the binary to log the targets of all indirect call or jump instructions for us automatically.

5.2.1 Instrumentation

This is a technique that injects instrumentation code into a binary to collect run-time information. It might just inject some `printf` functions to say when a function is entered, or inject things like a loop counter that is output when the loop increments. NOTE: it does not modify the semantics of the program.

Instrumentation is useful when we are profiling for compiler optimisation or performance profiling. It is also useful for Bug detection or vulnerability identification or even exploit generation. It is also used for architectural research, using both processor and cache simulation, and trace collection.

Static instrumentation is instrumentation performed before runtime. This comes in three flavours:

- **Source code instrumentation** — instrument source programs
- **IR instrumentation** — instrument compiler-generated (like LLVM)
- **Binary instrumentation** — Instrument executables directly by inserting additional assembly instructions.

Dynamic binary instrumentation is performed just after runtime (just in time — JIT). We use binary instrumentation because libraries are a big pain for source or IR level instrumentation. It also easily handles multi-lingual programs. Additionally, worms and viruses are rarely provided with source code.

- Pros:
  - No need to recompile or relink
  - Discovers code at runtime
  - Handles dynamically generated code
  - Attaches to running processes
- Cons:
  - Usually higher performance overhead
  - Requires a framework which can be detected by malware

5.2.2 Pin

Intel Pin is a dynamic binary instrumentation tool. It supports both Linux and Windows executables for x86, x86_64 and IA-64 architectures. It allows a tool to insert arbitrary code in arbitrary places in the executable while the executable is running. This also makes it possible to attach pin to already running processes.

Pin allows the full examination of any x86 instruction, tracking function calls (including library and sys calls), and tracking application threads, among others.

5.3 Fuzzing

Fuzzing is a type of analysis. It's not like fuzzy matching, or fuzzy logic, it is something else. It tests to see if memory corruption bugs are actually *exploitable*, that is to say we can wrangle it for nefarious uses. This matters because the number of vulnerabilities per year is going up. This means that the system itself is less secure.

The process is to run a program on many *abnormal/malformed* inputs and look for unintended behaviour. An observable side effect is essential, and it should be scalable. The underlying assumption is that if the unintended behaviour is dependent on an input, an attacker can craft such an input to exploit a bug.

There are multiple types of fuzzing:

- Input based: mutational and generative
  - Mutate seed inputs to create new test inputs
    - Simple strategy is to randomly choose an offset and change the byte
    - Pros: Very easy to implement and low overhead
    - Cons: Highly structured inputs will become invalid quickly because it has low coverage.
  - Generation Based: Learn/create the format/model of the input and based on the learned model, generate new inputs
    - Well-known file formats

        * Pros: Highly effective for complex structured input parsing applications because it has high coverage
        * Cons: Expensive as models are not easy to learn or obtain
- Application based: black and white box
  - Black box: Only the interface is known
  - White box: Inner workings of the program are known. Application can be analysed and monitored (can use static and dynamic analysis). Normally means we have the source code
- Input Strategy: memory-less and evolutionary

The problem with traditional fuzzing is that if we use black-box and mutation, we are essentially aiming with luck. Think shit and walls. So, if we apply more heuristics to mutate better and learn good inputs, we can apply more analysis to understand the application behaviour. This is known as *smart/evolutionary fuzzing*:

- Recall: memory-less and evolutionary fuzzing
- Rather than just throwing inputs at the program, *evolve them*
- The underlying assumption is that inputs are parsed enough before going further deep in execution

This will take some time and resources, so the scalability may be affected; do we have access to the resources and the required time to get this.

The feedback to enable the fuzzing to evolve should be either

- Code-coverage based fuzzing
  - Most of the contemporary fuzzers are here
  - Uses code-coverage as the proxy metric for the effectiveness of a fuzzer
- Directed fuzzing
  - This method is not much explored
  - The idea is that there should be a way to find the destination and a sense of direction.

For smart code-coverage based fuzzers, it is important to have some knowledge about:

- Where to apply mutation (which offsets in input)
- What values to replace with
- How to avoid traps (paths leading to error handling code)

Symbex can be used in combination with fuzzing and it means *symbolic/concolic* execution. Unfortunately, Symbex is not very scalable because symbolic execution programs are very resource heavy. It does allow us to collect strings, however. We are able to enhance the scaling capabilities of symbex:

- Native execution, contrary to IR based execution in existing symbex tools
- Instruction-level symbolic execution
- Optimistic solving

VUzzer goes even further with more analysis. The main idea is to prioritise/deprioritise certain paths, as some can be difficult to execute because they are guarded by constraints (nested conditions). They also leverage the application's control and data-flow features to infer input properties. It combines both static and dynamic analysis along with heuristics to improve coverage.

There are still problems, though. We are unaware of whether offset is processed by the application, which is a waste of mutation time. Also, we don't know what or where to mutate, since different bugs have different triggering conditions.

Evaluating fuzzers is also pretty hard, because how do we measure efficiency?

- Could measure code-coverage, but what about directed fuzzers?
  - Also for binary only fuzzers, measuring code coverage is not that straight forward.
  - For Code based fuzzers, what about library code?
- Uniqueness of crashes
  - How do you differentiate between several crashes? Often, coredump does not have enough information
  - Root cause analysis (not much is there)

5.3.1 Fuzzing for Race Bugs

Applying fuzzing for data race bugs is really interesting (allegedly). Race conditions are pretty serious (we have looked at this before) so fuzzing for these is pretty important. Actually, RAZZER is a fuzzer that was used to find some new race condition bugs in Linux.

We can fuzz the scheduler to identify these. Existing approaches are:

- Identify shared objects
- An input that executes instructions involving shared objects
- Thread scheduler
  - Rather than letting OS decide, introducing a scheduler that can control the thread scheduling
  - Schedule threads with respect to different ordering

6 Software Defence Mechanisms

We will look at compiler and OS level protections. The **stack canary** (also called security cookies) are values added to binaries during compilation to avoid buffer overflow attacks. Another method is to use W$\hat{\text{X}}$ (write xor execute). ASLR (address space layout randomisation) is a technique that means it is harder to get solid memory addresses in the stack.

6.1 Canaries

We know from buffer overflow attacks that:

- At CALL, return address is saved on the stack
- The stack grows downwards
- Local buffers are allocated on the stack
- Return address is POPed into the EIP

- EIP can point to anywhere in the memory

We can protect against this saved return attack by:

- StackGuard protection
- Use a different prologue
  - Push a canary into the stack
  - It's a constant `0x000aff0d`
- Different Epilogue
  - Checks to see if the canary is still there. If it has been overwritten, we know that a buffer overflow has happened

This works for `strcpy` because the `0x00` will stop `strcpy` from copying any further. These canaries are called *terminator canaries*.

Unfortunately, this isn't perfect. Local variables that are located after (on top of) **buf** are not protected. Also, the saved frame pointer EBP can be altered.

StackShield is another protection method. It saves return addresses in an alternate memory space named **retarray**. Two gloabl variables are used: **rettop**, initialised on startup and is the address in memory where retarray ends. The other variable is **retptr** and is the address where the next clone is to be saved.

The function prologue ensures the return address is copied from the stack to **retarray** and **retptr** is incremented. The epilogue ensures the saved clone RET is retrieved and is checked with the RET from the stack. If it has changed, we can either overwrite the changed one, or just exit.

With newer versions of GCC, there is a new stack layout:

- Function params
- Function return addresses
- Frame pointer
- Cookie
- Locally declared variables and buffers
- Callee save registers

6.2 Writeable xor Executable

The basic idea is that memory pages have permission to be either writeable or executable but not both. It protects against attacks that rely on code execution on data segments, and is a subclass of DEP (data execution protection).

6.3 Address Space Layout Randomisation

With ASLR, the predictability of memory addresses is reduced by randomising the address space layout for each instantiation of a program. Therefore, heap, stack, bss, and text section of the program get different addresses every time.

6.4 Control Flow Integrity

This is a strong attack mitigation technique. It restricts the control-flow of an application to *valid* execution traces. The assumption is that any attack deviates from the intended execution (which is the set of precomputed or static states). At runtime, if an invalid state is detected, an alert is raised, usually terminating the application. It is not the same as bug detection because other runtime monitoring techniques like sanitisers target development setting, detecting violations when testing the program (for example, during fuzzing). CFI on the other hand is an active defence mechanism to mitigate an ongoing attack.

CFI detects control-flow hijacking attacks by limiting the targets of control-flow transfers, and consists of two abstract components:

- Static analysis component that recovers the CFG of the application (at different levels of precision)
- The dynamic enforcement mechanism that restricts control flows according to the generated CFG. IT involves instrumentation (discussed earlier)

Control flow can be categorised in two ways:

- Unconditional and conditional jumps
- Direct (target known at compile time) and indirect (target is known at runtime)

Due to the write integrity of the code section, direct transfers are protected, so it is the indirect control-flow that we want to protect.

6.4.1 Types of indirect control flows

Forward edge transfers direct code forward to a new location and are used in indirect jump and indirect call instructions (such as `jmp *rax` or `callq *rbx`). **Backwards-edge transfers** are used to return to a location that was used in a forward-edge earlier (such as returning from a function call through a return instruction). Therefore, we have both forward and backwards-edge CFI.

6.4.2 Constructing CFG

Depending on the precision, a combination of static and dynamic CFG construction can be used. Forward edge over-approximates the targets of the indirect transfer, while indirect function calls are complex:

- Approximates based on function prototypes
- Different CFI mechanisms use different forms of type equality, so any valid function or functions with the same arity or functions with teh same signature. At runtime, any function with matching signature is allowed.

Function matching can be further enhanced by looking at **address-taken** functions. These are functions whose address is calculated and assigned. There are more complex approaches to enhance

the precision such as path sensitive computations.

6.4.3 Runtime enforcement

For forward-edge transfers, the code is often instrumented with some form of equivalence check — only valid targets are allowed. Each callsite and function entry-point is instrumented to check this equivalence (known as trampoline code). Backwards-edge transfers are harder to model as returns can come from any valid callsite. Shadow stacks are used to enforce the recent caller.

7 Hardware Vulnerabilities

We like to pretend that everything is digital and forget that hardware actually exists and can be a little bit analog. The bugs that we get from hardware can have some serious consequences for programmers.

7.1 DRAM — Rowhammer

When we access memory, we activate the row in the bank by letting the capacitor discharge into an active memory buffer. We have to actively refresh the data stored in the capacitors. Unfortunately, electronic engineering is messy:

- Capacitors are leaky!
- Current in wires can induce current in other nearby wires.
- 1s and 0s aren't really charged on uncharged capacitors, it's whether the capacitor is releasing more/less than a threshold voltage.

This used to be fine because electronic components were really large, but now that they're small, it's more of a problem. This leads to a bug in DRAM chips that has been known since 2010 called **Rowhammer**. The basic idea is that if you repeatedly charge and discharge a row in DRAM really quickly, it can sometimes cause errors in nearby rows. Manufacturers know about it, but it is not really documented since it is seen as a *reliability* issue. Cached memory largely fixes this, too. It was also discovered that you can flip bits in memory without even accessing them.

Rowhammer looks like this (in assembly)

```
code1a:
    mov (x), %eax
    mov (y), %ebx
    clflush (X)
    clflush (Y)
    mfence
    jmp code1a
```

Say X was in row 1 and Y in row 4, if you load X into the active buffer and Y into the b buffer really really fast, you'll get errors in rows 2 and 3. Some interesting things were discovered about this:

- Bit flips are *consistent*:
  - You might not know which bit will flip, but it will be the same bit
  - Same hardware generally flips the same bits
- Double sided row hammering makes flipping bits *really likely*
  - Even in chips thought to be resistant

If we can get a variable held in two different rows in one bank, we can induce a random, but consistent single bit error in surrounding rows, but it is hard to know how virtual memory translates to *real memory locations*. So, there will be an element of getting lucky, although we only need to get lucky once, and we can increase this luck by running many things at once.

This is cool and all, but we're just producing single bit errors, so what is the use of this?

7.1.1 NaCL

NaCL is a sandbox for C/C++ code that aims to make things safe. The sandbox runs with privileges and checks whether a chunk of code is safe to run. If so, it loads into memory aligned on 32B boundaries. If we can corrupt this, we can run unsafe hidden code.

The useful thing about x86 architecture is that there is no requirement for aligning instructions. Different instructions have different lengths, and some have multiple lengths!

The code section is readable by loaded processes, so we can spot when the rowhammering has been successful.

Attack method:

- Load a sequence of safe code that have unsafe instruction-sequences at 1-bit different offsets
- Rowhammer NaCL's code loading code (load NaCL into memory a lot and hope we get lucky)
- Either, we get an invalid instruction sequence (crash)
- Or we Rowhammer the kernels memory accidentally (crash again)
- Or we get lucky!

7.1.2 Preventing Rowhammer

We could buy better RAM, but how do you tell?

ECC RAM is an option that can fix single bit errors and reboot on 2 bit errors (but 3 bit errors are exploitable), but it is slower and more expensive.

7.2 Speculative Execution

Everyone remembers the fetch decode execute and write back cycle. This is how CPUs execute programs, and this cycle takes time (for each instruction, there are 4 things to do). On modern CPUs, we are able to parallelise (*pipeline*) the whole process. As one instruction has finished fetching, the

next is fetched while the first decodes. And then fetch the next as the first executes and the second decodes.

This works well so long as no instruction depends on the results of a prior one.

It becomes a little bit sticky after a conditional jump: do you know whether the program will take the branch? You could either block until you know for sure, or guess (and we guess).

If we correctly predict the branch, then great. If we get it wrong, this is fine *as long as you catch it before writeback*. There are a bunch of algorithms for this, such as 'always assume no', 'backwards taken, forwards not', or 'programmer hints'.

Modern branch prediction is much more complex, and a lot of the speed in computing comes from this prediction. The basic idea is that it keeps a record of what the branch has done before and predicts on the basis of that. It's not very well documented, varies chip by chip and is pretty obscure.

There are some unanswered questions here: What happens if executing an instruction speculatively (based on branch prediction) causes data to be loaded into cache? If so, should you flush the cache on branch misprediction?

7.3 Meltdown and Spectre

This is a family of real world side channel attacks based on specualtive execution. Meltdown *melts down* security barriers and spectre makes speculative execution scary. The attacks allow us to leak secure memory (in keys) and affects nearly all OSs and processor architectures. We do have some **software** mitigations available now, but they come at a considerable cost.

7.3.1 Spectre

Spectre really just looks like these two lines:

```
if (x < array1_size)
    y = array2[array1[x] * 4096]
```

`array1` is a pointer to some region of memory. We are allowed to access `array1_size` bytes of it. x is controlled by an attacker and 4096 is the size of a cache page.

Now, `if(x < array1_size)` shoudl fail, but it is going to take some time to get through the CPU pipeline. We can trick the branch predictor into guessing that the branch will be taken (make it succeed repeatedly beforehand) and then the second line will be executed speculatively.

`array2[array1[x] * 4096` is going to cause a page miss, so a new page will be loaded into the CPU's cache, and that page will depend on the value of `array1[x]` which would normally trigger a segmentation fault which happens during the last stage of a CPU's pipeline.

But, the exception won't be thrown because the branch predictor will catch up so the if statement should never have been taken, so all of the registers, exception flags, memory writes will be rolled

back.

However, the CPU's cache is NOT flushed. So, the currently cached memory page is dependent on whatever was at that illegally accessed memory address. So, if you were able to time accessing each page of memory, you could find out which page was quick to access adn that would leak the value or `array1[x]`.

You can also trigger this with JS. So you could actually host a webpage on a shared server, dump all of memory for every process and user on there, and you can leak every key in memory.

7.3.2 Mitigations

Some software mitigations we can do are at the OS levels:

- see `/sys/devices/system/cpu/vulnerabilities` on Linux
- Not perfect, but makes it **much** harder to exploit

But on an Intel CPU hyperthreading (SMT) makes exploiting **much** easier. Hyperthreading pretends that each core is essentially two cores (parallelisation is doubled). Hyperthreading should really be disabled, which is a massive shame because the performance cost of disabling this can be up to 25%.