Systems and Software Security

Josh Felmeden

November 16, 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.

6 Hardware Security

6.1 Rootkit

A rootkit is a malicious program that gives an attacker a permanent root access to a system, while simultaneously hiding its presence. It an steal information or allow remote code execution.

Typically, the attacker steps are as follows:

- Initial intrusion
- Open remote access
- Privilege escalation
- Download the malicious payload
- Install rootkit
- Perform malicious action on command

On a high level, the kernel space is where hardware can be controlled. User space interacts with the kernel which in turn interacts with the hardware. Between the kernel space and the hardware, we have the *virtual layer* where virtual machines are optionally managed. Our goal is to hide the rootkit in the *kernel space*.

```
int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);
```

The system call `getdents()` reads several *linux_dirent* structures from the directory referred to by the open file descriptor `fd` into the buffer pointed to by `dirp`. The argument `count` specifies the size of that buffer.

We are able to modify the returned value of the linux command `ls` to exclude our malicious code using `getdents()`. We can also modify the behaviour of anything that could reveal the presence of the malware and gives an easy means to obtain root privileges.

There are approximately three techniques:

- Modify the kernel code
- Hooking modify where certain functions point to
- Modify data structure (such as active process lists).

There are also multiple types of rootkit:

- Application
- Kernel
- Virtualised
- Bootloader
- Hardware and firmware

6.2 Attack Surface and Trusted Computing Base

The **attack surface** is all the possible ways for an attacker to compromise a system:

- Users
- Network
- Operating systems
- Software
- Hardware
- etc.

We will be looking at how to *limit* the attack surface. One such method is **sandboxing**. This is the process of isolating certain applications that might not be from a trusted source, where even if the code is malicious, we limit the spread of the code.

We want to define our sandbox with the following parameters:

- **Security**: Goal vs adversary
- **Policy**: Goal you want to achieve
  - Confidentiality
  - Integrity
  - Availability
- **Threat model**: assumption about the adversary
  - Reasonable assumptions
  - Attacker omnipotent nothing can be done
  - But also need to not be too weak

- **Mechanism**: how the policy is implemented

The **trusted computing base** (TCB) is the part of the system that we trust in order to achieve the objective.

6.3 Trusted Platform Module

The **trusted platform module** (TPM) is:

- Trusted computing group
  - Microsoft, Intel, IBM etc
- Promoting standard for more trusted computing
  - Additional chip on the motherboard
  - Called TPM
- Used for
  - Disk encryption
  - **System integrity**
  - Password protection
  - and more

We can achieve trust if we can verify the system has booted correctly. We assume the PC hardware has not been modified, and we need to monitor the boot process:

- Initial boot measure by the core root of trust (CRTM)
- Hash the BIOS, store results in TPM, start the BIOS
- Let BIOS do its job, load next stage, hash it, store in TPM, etc.

To ensure that the boot has loaded correctly, we follow those steps to authenticate the boot.

The TPM registers are:

- Platform configuration registers (PCRs)
- A PCR hold the summary of a series of values
  - Not the entire chain of hash
  - The chain could be infinite
- A PCR register is extended
  - PCR = HASH
  - Shielded TPM location
  - Measurements are provided by software

For TPM, we rely on **remote attestation**:

1. An untrusted prover P and a trusted verifier V.
2. V knows P expected memory content.
3. V sends challenge with a nonce to P.
4. P computes a measurement.
5. V verifies the measurement.

Remote attestation can either give:

- Positive result
  - Correct memory content
  - Good device
- Negative result
  - Malfunctioning/malicious device
- No response
  - Malfunctioning/malicious device

The PCR cannot be modified and only reset at reboot. TPM contains a key used to sign the attestation. The verifier will verify both the TPM certificate and the PCRs. The attestation contains the PCRs value and the sign of the PCRs against the nonce.

TPM and remote attestation don't need to stop at the OS, they can realistically attest pretty much everything.

There are some limitations of this, however. It is only able to verify static information. For long running applications, it might be necessary to reboot the system to do a sensitive operation. Runtime status of a device is not known, so an attacker can compromise a system during execution. Finally, reboot might not be sufficient, for example, jailbreaking an iPhone circumnavigates the 'safe boot', which should only allow secure code to execute.

6.4 Intel Secure Guard Extension (SGX)

Intel SGX is a secure extension to the operating system. Returning to rootkits, we know that attackers can compromise pretty much everything. To avoid this, we can execute code in its own enclave, and this is where SGX hardware comes in. (NOTE: there are some vulnerabilities)

The main idea is that we run an application within some isolation unit so it cannot be affected by the OS. Don't trust the OS or the hypervisor, and only the hardware is trusted, thereby reducing the attack surface to only the virtual later, or certain layers of the app.

SGX prevents memory snooping attacks since security boundary is the CPU package. Data is encrypted outside the CPU, meaning it is not possible to see the data passing between hardware.

Each *enclave* has its own code and data. There are controlled entry points meaning that we can only enter enclave code at a specific point.

The SGX application flow is as follows:

1. Define and partition application into trusted and untrusted parts
2. App create enclave
3. Trusted function is called
4. Code in enclave process some secret
5. Trusted function returns
6. App continues as normal

6.5 ARM Trustzone

ARM Trustzone is similar to Intel SGX, but it has a secure zone that has trusted OS, and trusted application running alongside other, untrusted software. Within the untrusted zone, there will be trusted drivers that are given information from the dispatcher, run by a secure monitor that runs below all other software.