Systems and Software Security

Josh Felmeden

October 4, 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*.