Lecture 1: Interesting bugs and where to find them

Joseph Hallett

September 4, 2024

Welcome to Hacking

Welcome to Hacking

…er, no wait I'm not alowed to call this unit that.

- Apparently they're worried we might look like we're doing something illegal
- We could call it *Ethical Hacking* if we liked…

One problem with that…

Welcome to ~~Hacking~~ Systems and Software Security

- Nevertheless, what we're going to be studying is *very much* hacking.
- Combination of labs and lectures
  - Get a high level overview of a technique, then get an opportunity to do it for real.
  - Chance to ask questions and chat with us.
- Each week there will be problem sheets
  - The problems in them **exactly** match the style of the exam questions
  - There are solutions…
  - **BUT** I'd rather you just came and chatted with us about what your answers were in the lab
  - There usually isn't a right answer…

Assessment

For the exam portion:

- It'll be a mixture of long and short essay style questions
- Past exam available, questions each week are same style

Open book

- We don't care what form of notes you bring in
- We don't care how many notes you bring in
- Lecture 1 slide 4 says you can bring in whatever you like.

For those of you doing the coursework:

- It'll be take a technique from the labs (ROP) and implement a general purpose tool to use it to automate an attack
- Individual or group work
- Marked according to how much you get done and how many people you had working on it

There will also be a mid-term test

- We're going to give you a program you've not seen before
- You have to exploit it (stack smashing and shellcode)
- We watch you do it
- If you **can't do it** theres no way you can complete the coursework

Reading and Homework

This is a **Masters** level unit!

- I **really** like reading
- There will be **oodles** of reading

The reading is optional **but**:

- It will probably give you good examples and context to illustrate exam answers with ;-)
- It will definitely help explain ideas in more detail
- You may prefer their explanations to mine

I'll also try and link to other people's explanations of things:

- In particular *LiveOverflow* on YouTube
- Everyone learns differently…
- …if you don't like my style, use someone else's notes

Security

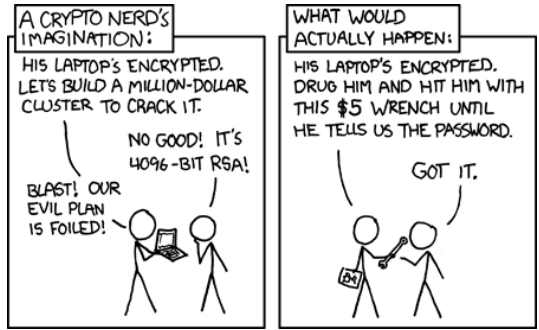


Figure: XKCD #538: Security

(In our case the spanner would usually be a *keylogger*).

Lisp

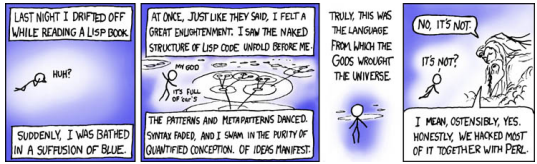


Figure: XKCD #224: Lisp

Assumptions

Software is built by building on other software (and hardware).

- Abstracting stuff away into libraries means you don't need to know how the library works.

But *sometimes* the way people assume abstractions work, and how they actually work get misaligned.

- This leads to opportunities for bugs and crashes!

The language spec might say *undefined behaviour* **in theory**…

- But sometimes that behaviour is very much defined **in practice**.

Today's lecture

Three classic security bugs:

- What the assumptions are behind them
- How those assumptions get broken
- How we exploit them
- How we fix them (the important bit)

The next two labs will focus on exploiting two of the bugs, in undefended systems

- The lab after will focus on doing it in a modern defended system.

Bug #1: Race conditions

Computers can do more than one thing at once and the order is important.

```
void increment(int *n) {
  int temp;
  temp = *n; // Get the value pointed to by n
  temp += 1; // Add one to it
  *n = temp; // And write it back
}
```

What happens if n is dereferenced again before the increment function has had time to complete?

- We will lose one (or more of the increments).
- Well understood *correctness* issue.
- Fix by implementing synchronous blocks or locking.

Access and open

System calls for testing the permissions of a file and then opening them up.
From the `access 2` manual page:

> *The `access()` system call checks the accessibility of the file named by the **path** argument for the access permissions indicated by the **mode** argument. The value of the **mode** is either the bitwise-inclusive OR of the access permission to be checked `R_OK` for read permission, `W_OK` for write permission, and `X_OK` for execute/search permission), or the existence test (`F_OK`).*

So what does this look like?

```
if (access("/tmp/X", W_OK)) {
  f = open("/tmp/X");
  write_to_file(f);
} else {
  printf("Nope.\n");
}
```

Suppose this code is being executed by root, and that `/tmp/X` is owned by a non-root user? Can we do similar tricks to get root to write to an arbitrary file?

Oh dear…

```
ln -s /etc/password /tmp/X
```

What happens if we swap out the temporary file with a link to a file we wouldn't normally have access to after the access control check is done, but before the open has started?

- Bad things (for the system admin).

The assumption is that nothing can happen between *check* and *use* of data; but this assumption isn't always right.

- Be *really* careful when splitting things into steps.
- Some static analysis tools can spot things.

You were warned…

SECURITY CONSIDERATIONS

The result of `access()` should not be used to make an actual access control decision, since its response, even if correct at the moment it is formed, may be outdated at the time you act on it. `access()` results should only be used to pre-flight, such as when configuring user interface elements or for optimization purposes. The actual access control decision should be made by attempting to execute the relevant system call while holding the applicable credentials, and properly handling any resulting errors; and this must be done even though `access()` may have predicted success.

Additionally, set-user-ID and set-group-ID applications should restore the effective user or group ID, and perform actions directly rather than use `access()` to simulate access checks for the real user or group ID.

Bug #2: Buffer overflow

In C if you go over the end of an array you'll crash.

- A *buffer overflow*!

```
#include <string.h>
int main(void) {
  char *string = "One␣small␣step␣for␣man";
  strcpy(string, "One␣giant␣leap␣for␣a␣class␣full␣of␣prospective␣hackers");
  return 0;
}
```

Bus error: 10

But why does your program crash?

- How does the program/OS know that something has gone wrong?

Segmentation faults

Memory in a binary get split into *sections*…

- If you attempt to write in memory without having the permission the MMU will trigger an exception to the OS and the program will crash with a segfault.
  If you dump the sections of a binary using R2

```
rabin2 -S /tmp/compiled-example-from-last-slide
```

nth	paddr	size	vaddr	vsize	perm	name
0	0x00003f14	0x48	0x100003f14	0x48	-r-x	0.__TEXT.__text
1	0x00003f5c	0xc	0x100003f5c	~0xc~	-r-x	1.__TEXT.__stubs
2	0x00003f68	0x4e	0x100003f68	0x4e	-r-x	2.__TEXT.__cstring
3	0x00003fb8	0x48	0x100003fb8	0x48	-r-x	3.__TEXT.__unwind_info
4	0x00004000	0x8	0x100004000	~0x8~	-rw-	4.__DATA_CONST.__got

But sometimes we get away with it right?

Sometimes if you don't go too far off the end of an array the program crash it just continues

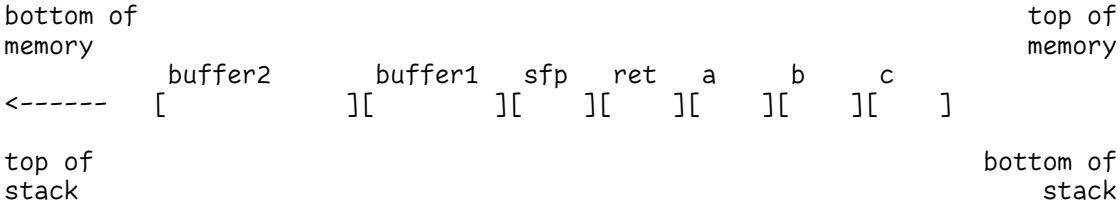
- So what happens *before* we trigger the fault?

So how do functions work?

```
void function(int a,
              int b,
              int c) {
  char buffer1[5];
  char buffer2[10];
}

void main() {
  function(1, 2, 3);
}
```

```
function:
        push ebp
        mov ebp,esp
        sub esp,0x20
        leave
        ret
main:
        push ebp
        mov ebp,esp
        push 3
        push 2
        push 1
        call function
        leave
        ret
```

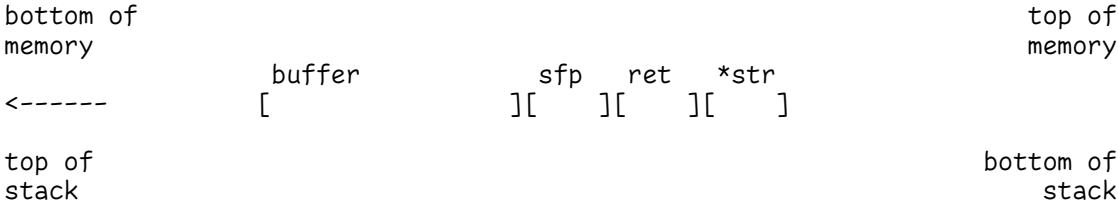
So what does this look like in memory?



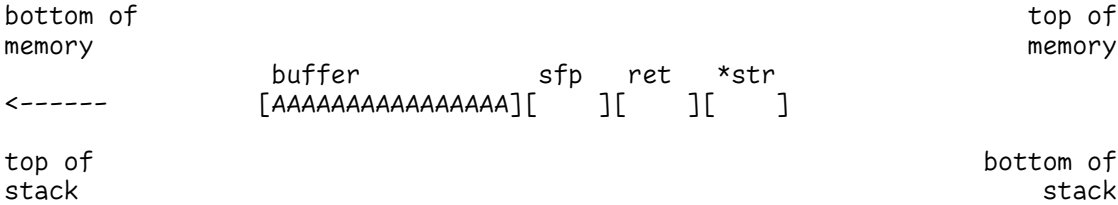
Lets try something a bit more complex

```
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);
}
```

Now memory will look like

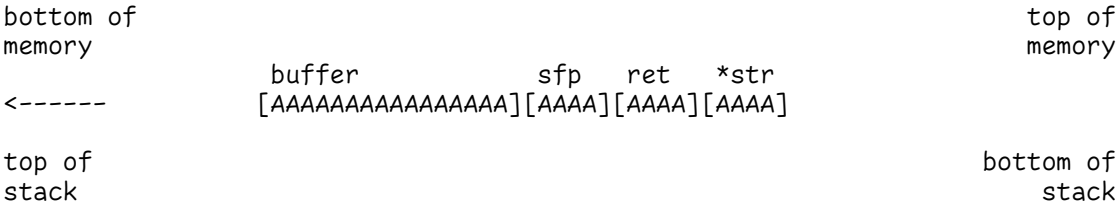


As memory fills…



First the buffer will be filled with ~A~s…

Little by little...



Then whatever else is on the stack will be overwritten...

And eventually a crash…

```
bottom of                                                          top of
memory                                                             memory
                  buffer             sfp    ret    *str
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][AAAA]AAAAAAAAAAAAAAAAAAAA...

top of                                                             bottom of
stack                                                                stack
```

And then whatever else until the segfault happens…

- but what if we stop before the segfault happens?

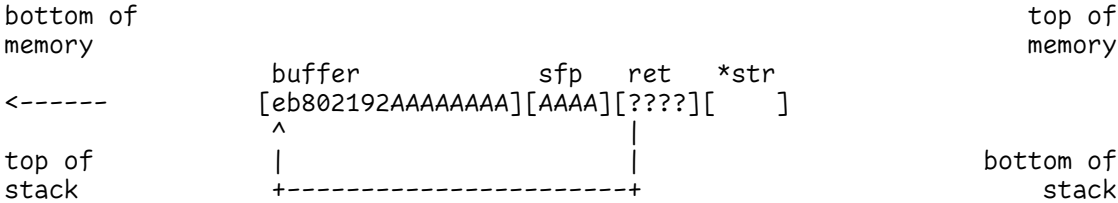
Return to a crash

```
bottom of                                                          top of
memory                                                             memory
                    buffer              sfp   ret   *str
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][    ]

top of                                                             bottom of
stack                                                                stack
```

Let's say we stop here: what is going to happen next?

- The return address will be loaded into memory and we'll start executing from `0x41414141` (`'A'` in ASCII is byte `0x41`).
- Which is *likely* to be a junk address and will crash the program from a nonsensical instruction/non-executable memory.

But what if it isn't?



Suppose instead of returning to 0x41414141 we aim to return to something a little more useful

- Stack addresses are usually predictable-ish

Suppose we stick into our buffer not just a series of letters, but something that corresponds to program code?

- We could trick the computer into running arbitrary programs.
- This is really bad… (arbitrary code execution).

Faulty assumptions

You can't assume that a program will crash just because that's what its supposed to do.

- Sometimes undefined behaviour is pretty defined in practice right up until the crash.

How do we stop it?

- Make addresses less predictable? If its hard to guess where in the stack you are you're less likely to be able to calculate it cleanly (ASLR).
- Stick a canary in the stack and check it hasn't been overwritten before returning (stack canaries).
- Use a research grade CPU architecture like CHERI that doesn't allow you to abuse pointers like this (come chat with me in the labs!)

Bug #3: Format strings

…or why is it *sometimes* fun to set your phone's name to %08x
What happens if I compile this:

```
printf("Hello␣%s");
```

Oh but its only a *warning*...

```
warning: more '%' conversions than data arguments
printf("Hello %s");
               ^
```

The program gives a warning (but still compiles it).

- If I run it it'll *probably* crash with a segfault when you dereference that pointer for the %s specifier.

There's that word again

Probably.

- Oh it'll *probably* crash.
- But we are enterprising and devious!
- Can we get it to do something useful?

Here's a silly program!

```
int main(int argc, char *argv[]) {
  printf("This␣program␣is␣called:␣");
  printf(argv[0]); // What! You never heard of %s?
  printf("\n");
  return 0;
}
```

It just prints the name of our program, the compiler warns it may be insecure but surely not?

```
warning: format string is not a string literal (potentially insecure)
printf(argv[0]); // What! You never heard of %s?
       ^~~~~~
```

Look I don't come up with the naming conventions

What happens if we name our program something silly… like `%08x-%08x-%08x-%08x`

```
$ ./%08x-%08x-%08x-%08x
This program is called: ./000120a8-00000000-118a0041-4f6d0188
```

Well that's not right… what on earth is going on?

- The answer is that `printf` is assuming you know what you're doing.
- It doesn't know that the arguments weren't passed, and is assuming you know what you're doing and will put the correct values in to where it is expecting them.
- This gives us a mechanism for dumping whats on the stack (and potentially registers) and leaking information.

It's a good job the `printf` functions just print stuff, right?

No writing here!

- So its not going to lead to arbitrary code execution!
- Right?
- *…right…?*

Tabular output

Say you want to print an address book on an old teletype and you're too clever for your own good.
Something like:

```
Joseph Hallett Office 3.36
               Merchant Venturers Building
               Opposite the stairs

Sana Belguith Just down the corridor from 3.36
              Also Merchant Venturers Building
              Near IT
```

How do you ensure that the addresses are aligned?

- A sensible person would use 'strlen' after printing the name.
- But an efficient programmer, on the other hand…

…But an efficient programmer

```
unsigned int offset;
printf("%s%n%s\n", name, &offset, address[0]);
while(++address) {
  for (int i = 0; i < offset; i++)
    putchar(' ');
  printf(" %s\n", address);
}
```

Saves a whole single call to `strlen`!

- What a wonderful optimization!
- Whole nanoseconds of time eliminated!

%n

If you dive into the bowels of your C programmers manual in `man 3 printf` you'll find this beauty when it describes all the format specifiers:

> ***n***
> *The number of characters written so far is stored into the integer indicated by the* `int *` *(or variant) pointer argument. No argument is converted. The format argument must be in write-protected memory if this specifier is used; see* ***SECURITY CONSIDERATIONS*** *below.*

Nothing worrying there eh!?

Security considerations?

> *%n can be used to write arbitrary data to potentially carefully-selected addresses. Programmers are therefore strongly advised to never pass untrusted strings as the format argument, as an attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole.*

Good thing we're not going careful to select our addresses, and we know that nothing useful will *ever* end up on the stack.

- Especially not a return address!
- Or anything else we might want to corrupt…

How do we fix this?

So this stupidity is in the C library which means this behaviour is *standard*.

- Worse than that, it's mandated by the standard which means if you want to have a standards-conformant compiler…

Almost all systems have decided that a *truly* standards compliant compiler isn't worth it.

Windows	Does not implement %n for `printf`.
OpenBSD	Crashes your program and sends an email to the system admin telling them all about the sort of format strings you use.
MacOS	Removed the %n format specifier (1989s defences for 2021!).
Linux	…is standards compliant.

Don't assume it is safe to ignore warnings!

- …and if when connecting your phone to a car or bluetooth speaker it calls it *Connected to 028ffee1!*…

Smashing the stack for fun and profit

A lot of this is covered in various papers.

From 1996 http://phrack.org/issues/49/1.html

From 2021 http://phrack.org/issues/70/13.html

We're going to spend a lab or two playing with these last two bugs

- But I'm going to turn the defences off, so its easy and possible for you to hack like its 1999!
- And then I'll turn them back on then once we've covered ROP and we'll start doing some state of the art stuff!

What we covered

- An introduction to how faulty assumptions lead to three different types of classic software bugs.
  - *Race conditions, buffer overflows, and format string vulnerabilities.*
- Gave an overview of how to exploit them to get *privilege escalation* and *arbitrary code execution*.
- Gave an introduction about how to defend against them.

Next time

- Lab 1 will be a refresher course in assembly
  - (If you're already very happy programming in assembly feel free to breeze through it and start the next one…)
  - (Or y'know help those who thought their assembly days were behind them with Intro to Computer Architecture…)
- Then labs to practice what we learned
- Next lecture: heap overflows and (or everything you didn't want to know about how `malloc` works).