Today

More memory issues: buffer overflows

- stack smashing
- heap spraying

Mitigations

Buffers

Useful things!

Demo: sum.c

- a program that loads and sums some integers
- example: numbers.dat
- (also see Makefile)

It's been said that most of computing is a matter of transforming things from one representation into another so that we can do computation (and then, likely, to transform those results into another representation!). In order to do that, we often need to store information somewhere... like in a buffer!

This example program loads integers from a file and adds their values together. Some things to note as we walk through this example:

- low-level file I/O functions from the C standard library
- "hex" tools (binary viewing / manipulation using hex representations)
- endianness

Buffer problems

Q: what if we load too much data?

For example:

- big.dat
- error.dat

If we load too much data into our buffer, we _____. The consequences of this depend on _____!

Buffer overflow

Without bounds checks... memory corruption!

What is the consequence of this corruption?

- depends on **how much** we overflow the buffer
- depends **where** the overflowed buffer is

When we attempt to process big.dat, we see a `SIGSEGV` in the C standard libary's `memcpy` function. This is because we are copying in so much data that we walk right out of _____ _____. We can investigate this with a debugger, examining the _____ _____ to see who called what.

When we attempt to process error.dat, we still see a `SIGSEGV`, but it's for a much more interesting reason. When we try to investigate *this* case with a debugger, we can't even see _____ _____! Why is this?

Loading 0x10 (i.e., 16) integers from the file error.dat *overflows* the eight-integer (i.e., 32 B) buffer that we are loading into. This overwrites whatever comes after the buffer. The significance of that depends on where the buffer we're loading into is located!

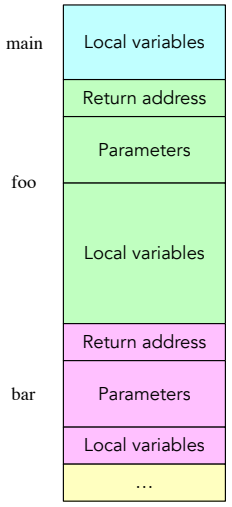
The call stack

Given what we know about stacks...

If we overflow a local variable...

What happens?

error.dat



Loading 16 integers from the file error.dat overwrites whatever comes after the buffer we're loading into. In the case of a local variable stored on the stack, this might be _____ _____, but it might also be any of the things that live on the stack between functions' local variables. For example, it might be a _____!

Stack smashing

Changing return addresses can cause crashes

But can we get even more creative?

malice.dat
(compiled from sh.s: assembly for FreeBSD, Linux and OpenBSD)
This is known as "shellcode", as it "pops a shell"

See: "Smashing The Stack For Fun And Profit" by "Aleph One"

We can *always* get more creative. 🙂

"Popping a shell" can be a beachhead in an attack on a real system, as an attacker can then execute arbitrary commands. It can also be a demonstration that the attacker *could* execute arbitrary commands if they wanted to.

What just happened?

Payload

- loaded attacker-provided code into memory
- all ready to be executed by...

Control-flow highjacking

- in this case, overwriting the return address (two birds, one stone)
- in other cases: other attacks!

Prevention

How can we prevent stack smashing?

- write perfect software!
- *memory-safe* languages (**partial** answer)

We will talk about memory safety in the next couple of lectures. We will also explore some of the tools that we used in today's lecture in the upcoming labs!

Mitigations

How can we prevent/reduce stack smashing?

- stack canaries: `-fstack-protector`
- non-executable stacks (we needed `-z execstack` to demo!)
- `W^X`: memory regions writable **or** executable (limitations?)
- ASLR: address space layout randomization (more later)

... and more to follow

A *stack canary*, like a canary in a coal mine (fun picture here), is something that can be checked to see if conditions are too dangerous to continue normal operations. In the case of a canary, it would faint from carbon dioxide before humans would, sending a signal that the mine wasn't safe. In the case of a stack, _____ can be written to the stack in between functions' allocations. Code is inserted to check this "canary" value _____ _____ to ensure that _____.

Marking memory as *non-executable* is something that wasn't possible on 32-bit x86 computers, but *is* possible on _____. This functionality can be used to prevent executable stacks (always a good idea!) and/or a full `W^X` policy.

The attacker strikes back

Guessing precise addresses is hard

`nop` sleds, relative addressing

Shellcode authors avoid zeroes (why?)

Is shellcode easy to spot? See: English shellcode*

* "English Shellcode", Mason, Small, Monrose and MacManus, in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009. DOI: 10.1145/1653662.1653725

For this demo to work, I had to embed the stack address that would store the program counter. How did I do this? By _____ when I ran the program! That is not a very reproducible solution. In reality, it may not be possible to guess where a piece of data will land in a program's memory.

To deal with this difficulty, shellcode can include "nop sleds", which are long chains of `nop` instructions with shellcode at the end. If the program counter lands anywhere within the `nop` sled, it will "slide" all the way to the end and then execute the payload that's found there.

These kinds of payloads are often delivered via code that expects to read strings from somewhere (a file, the network, the user, etc.). When the target code receives a string and passes to around to functions, etc., it's very likely to need to run functions like `strlen` to figure out how much data to pass, etc. So, a shellcode author needs to avoid zeros in their strings: if not, `strlen` will think that _____ and then the code will _____ _____!

You might think that shellcode would be easy to spot, but you can hide all kinds of things inside innocuous-looking content. Remember how, when we examined our malicious file with tools like `hexyl` or `xxd`, many of the characters in the shellcode were displayed like ordinary ASCII characters? A lot of instructions' opcodes _____!

Summary

Buffer overflows

Stack smashing

Heap spraying

Mitigations

... with more to follow!