Last time

We continued our 1st section: **Software Security**

- Buffer overflow fundamentals

By launching **Buffer overflows** and other memory safety vulnerabilities

This time

We will finish up **Buffer overflows** and other memory safety vulnerabilities

- Finish overflow attacks & other vulnerabilities
- Overflow defenses

By looking at **Code Injection, Defenses**
Buffer overflow: Recap

Can over-write other data ("AuthMe!")
Can over-write the program’s control flow (%eip)

```
char loc1[4];
gets(loc1);
strcpy(loc1, <user input>);
memcpy(loc1, <user input>);
etc.
```

Code injection
High-level idea

```c
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```

(1) Load my own code into memory

(2) Somehow get `%eip` to point to it

This is nontrivial

- Pulling off this attack requires getting a few things really right (and some things sorta right)
- Think about what is tricky about the attack
  * The key to defending it will be to make the hard parts really hard
Challenge 1

Loading code into memory

- It must be the machine code instructions (i.e., already compiled and ready to run)

- We have to be careful in how we construct it:
  - It can’t contain any all-zero bytes
    - Otherwise, sprintf / gets / scanf / … will stop copying
  - How could you write assembly to never contain a full zero byte?
  - It can’t make use of the loader (we’re injecting)
  - It can’t use the stack (we’re going to smash it)

What kind of code would we want to run?

- Goal: **full-purpose shell**
  - The code to launch a shell is called “shell code”
  - Nontrivial to launch it through injected code
    - No zeroes, can’t use the stack, no loader dependence
  - There are many out there
    - And competitions to see who can write the smallest

- Goal: **privilege escalation**
  - Ideally, they go from guest (or non-user) to root
Shellcode

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Privilege escalation

- Permissions later, but for now…
- Recall that each file has:
  - Permissions: read / write / execute
  - For each of: owner / group / everyone else
- Consider a service like `passwd`
  - Owned by root (and needs to do root-y things)
  - But you want any user to be able to run it
Effective userid

• Userid = the user who ran the process

• Effective userid = what is used to determine what access the process has

• Consider passwd: root owns it, but users can run it
  • getuid() will return you (real userid)
  • seteuid(0) to set the effective userid to root
    - It’s allowed to because root is the owner

• What is the potential attack?

**If you can get a root-owned process to run setuid(0)/seteuid(0), then you get root permissions**

Challenge 2

**Getting our injected code to run**

• All we can do is write to memory from buffer onward
  • With this alone we want to get it to jump to our code
  • We have to use whatever code is already running

![Buffer Diagram]

Thoughts?
Stack and functions: Summary

Calling function:
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you: %eip+something
3. Jump to the function’s address

Called function:
4. Push the old frame pointer onto the stack: %ebp
5. Set frame pointer %ebp to where the end of the stack is right now: %esp
6. Push local variables onto the stack; access them as offsets from %ebp

Returning function:
7. Reset the previous stack frame: %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%ebp)

Hijacking the saved %eip

But how do we know the address?
Hijacking the saved \texttt{\%eip}

What if we are wrong?

This is most likely data, so the CPU will panic (Invalid Instruction)

Challenge 3

Finding the return address

• If we don’t have access to the code, we don’t know how far the buffer is from the saved \texttt{\%ebp}

• One approach: just try a lot of different values!

• Worst case scenario: it’s a 32 (or 64) bit memory space, which means $2^{32}$ ($2^{64}$) possible answers

• But without address randomization:
  • The stack always starts from the same, \texttt{fixed address}
  • The stack will grow, but usually it doesn’t grow very deeply (unless the code is heavily recursive)
NOP Sled

**nop** is a single-byte instruction (just moves to the next instruction)

Jumping *anywhere* here will work

Now we improve our chances of guessing by a factor of #nops

---

Putting it all together

But it has to be *something*; we have to start writing wherever the input to `gets/etc.` begins.

But it has to be something; we have to start writing wherever the input to `gets/etc.` begins.
Project one due
Sep 26

Defenses
Recall our challenges

How can we make these even more difficult?

• Putting code into the memory (no zeroes)

• Getting %eip to point to our code (overwrite stored eip)

• Finding the return address (guess the raw addr)

Detecting overflows with **canaries**

Not the expected value: abort

What value should the canary have?
Canary values

From StackGuard [Wagle & Cowan]

1. Terminator canaries (CR, LF, NULL, -1)
   • Leverages the fact that scanf etc. don’t allow these

2. Random canaries
   • Write a new random value @ each process start
   • Save the real value somewhere in memory
   • Must write-protect the stored value

3. Random XOR canaries
   • Same as random canaries
   • But store canary XOR some control info, instead

Recall our challenges

How can we make these even more difficult?

• Putting code into the memory (no zeroes)
  • Option: Make this detectable with canaries

• Getting %eip to point to our code(overwrite stored eip)

• Finding the return address (guess the raw addr)
Recall our challenges

**How can we make these even more difficult?**

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries

- Getting %eip to point to our code (overwrite stored eip)
  - Non-executable stack doesn’t work so well

- Finding the return address (guess the raw addr)
Address Space Layout Randomization (ASLR)

- Basic idea: change the layout of the stack
- Slow to adopt
  - Linux in 2005
  - Vista in 2007 (off by default for compatibility with older software)
  - OS X in 2007 (for system libraries), 2011 for all apps
  - iOS 4.3 (2011)
  - Android 4.0
  - FreeBSD: no

How would you overcome this as an attacker?

Overflow defenses summary

- Putting code into the memory (no zeroes)
  - Option: Make this detectable with canaries
- Getting %eip to point to our code (overwrite stored eip)
  - Non-executable stack doesn’t work so well
- Finding the return address (guess the raw addr)
  - Address Space Layout Randomization (ASLR)
    - Many systems slow to adopt; also, how could you get around this?
- Good coding practices
Required reading:
“StackGuard: Simple Stack Smash Protection for GCC”

Optional reading:
“Basic Integer Overflows”
“Exploiting Format String Vulnerabilities”


Cat and mouse

- **Defense**: Make stack/heap non-executable to prevent injection of code
  - **Attack response**: Return to libc

- **Defense**: Hide the address of desired libc code or return address using ASLR
  - **Attack response**: Brute force search (for 32-bit systems)
    or information leak (format string vulnerability: later today)

- **Defense**: Avoid using libc code entirely and use code in the program text instead
  - **Attack response**: Construct needed functionality using return oriented programming (ROP)
Return oriented programming (ROP)

Return-oriented Programming

• Introduced by Hovav Shacham in 2007
  • *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, CCS’07

• Idea: rather than use a single (libc) function to run your shellcode, **string together pieces of existing code, called gadgets**, to do it instead

• Challenges
  • **Find the gadgets** you need
  • String them together
Approach

- Gadgets are instruction groups that end with `ret`

- Stack serves as the code
  - `$esp = program counter$
  - Gadgets invoked via `ret` instruction
  - Gadgets get their arguments via `pop`, etc.
    - Also on the stack

Simple example

Simple example

```
0x17f: pop %edx
      ret
```

```
mov %edx, 5
```

**goal: put 5 into edx**

`%edx | 5`

```
%esp %eax %ebx %edx
0xffffffff
```

**“program counter”**

```
0x00 ... 0x17f 5 next
gadget
Gadget
```

**“Instructions”**
**Code sequence (no ROP)**

- `%eax` 5
- `%ebx` 0x404

**Equivalent ROP sequence**

- `%eax` 5
- `%ebx` 0x404
Whence the gadgets?

- **How can we find gadgets to construct an exploit?**
  - Automate a search of the target binary for gadgets (look for `ret` instructions, work backwards)
    - Cf. https://github.com/0vercl0k/rp

- **Are there sufficient gadgets to do anything interesting?**
  - Yes: Shacham found that for significant codebases (e.g., libc), **gadgets are Turing complete**
  - Especially true on x86’s dense instruction set
  - Schwartz et al (USENIX Security ’11) have automated gadget shellcode creation, though not needing/requiring Turing completeness
Blind ROP

- **Defense:** Randomizing the location of the code (by compiling for position independence) on a 64-bit machine makes attacks very difficult
  - Recent, published attacks are often for 32-bit versions of executables

- **Attack response:** Blind ROP
  - If server restarts on a crash, but does not re-randomize:
    1. Read the stack to leak canaries and a return address
    2. Find gadgets (at run-time) to effect call to write
    3. Dump binary to find gadgets for shellcode

http://www.scs.stanford.edu/brop/

Defeat!

- The blind ROP team was able to completely automatically, only through remote interactions, develop a remote code exploit for nginx, a popular web server
  - The exploit was carried out on a 64-bit executable with full stack canaries and randomization

- Conclusion: **give an inch, and they take a mile?**

- Put another way: **Memory safety is really useful!**
```c
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}

void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}

void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```