Security and Exploit Mitigation

CMSC 421 - Spring 2016

Lawrence Sebald

Security is of Supreme Importance in Systems

- As we have seen in the past two classes, even with sophisticated security systems, small failures can snowball

- “You can break 20% of the security and [do] 100% of what [they] don’t want you to do”

- Attackers motivated by many different goals: stealing information, piracy, fame

- There is no fully secure system — any system that anyone has access to (remotely or locally) cannot be secured completely

What can be done?

- As we have seen, many sophisticated techniques have been proposed to deal with security in systems
- Encryption and code-signing are two good ideas
    - However, they do not alone make for a completely secure system
- A chain of trust is needed
    - But can often be broken, especially with physical access to hardware

So…

- A variety of techniques have been discussed to deal with security issues beyond the ideas of encryption and code signing — each with their own benefits and drawbacks
- We will discuss three today:
  - Buffer/stack guards
  - W^X
  - ASLR (and KASLR)

Buffer Overflows

- Buffer overflows are one of the most prevalent security flaws that are exploited by attackers today

- Essentially, they allow an attacker to craft a specific input that causes a write to a buffer to go beyond the bounds of what that buffer would normally hold

- Take the code on the next slide as an example…

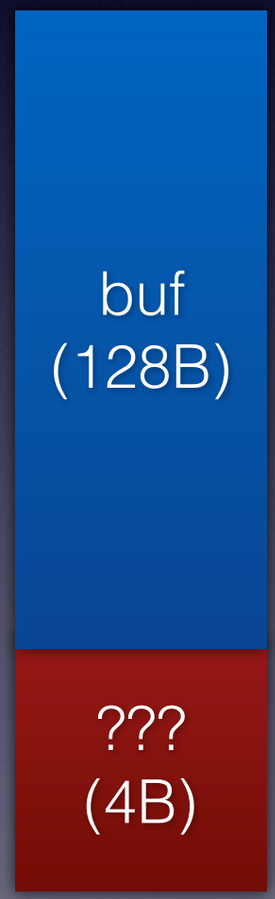
```
#include <string.h>

void magic(const char *input) {
    char buf[128];

    strcpy(buf, input);
    do_cool_stuff(buf);
}
```

```
_magic:
    sts.l   pr,@-r15
    mov.l   .L2,r0
    add     #-128,r15
    mov     r4,r5
    jsr     @r0
    mov     r15,r4
    mov.l   .L3,r0
    jsr     @r0
    mov     r15,r4
    add     #64,r15
    lds.l   @r15+,pr
    rts
    nop
.L4:
    .align 2
.L2:
    .long _strcpy
.L3:
    .long _do_cool_stuff
```

What happens if input is >128 bytes long?

Normal Stack Frame

Attacked Stack Frame
132B input



The attacker has overwritten the return address of the function magic()!

Buffer guards are one mitigation technique

- Canaries are widely deployed
  - Special value is written to the stack between the local data of the function and the data used for linkage
  - This value is checked on function exit
  - If the check fails, program is terminated
  - Canary values must be protected from disclosure to attackers — or they could craft their buffer overflow to include the canary!

Memory Protection

- When setting up pages for program use, various protection bits are set in the page table/TLB
- Usually amongst these are bits for whether the page should be readable, writeable, and/or executable
- Page protection bits are checked by each memory access, and a protection violation is generated if the access doesn’t comply with the flags set on the page
- Implemented in the mmap() and mprotect() system calls in *nix systems

W^X

- W^X means Write XOR Execute
- That is to say, enforce that no pages that are writable are ever executable as well
- This ensures that an attacker can’t fill memory buffers with code and use a buffer overflow on the stack to jump to it
    - The destination of the jump would be marked as non-executable, as the attacker was able to write to it, thus the program would be terminated

W^X

- Modify mprotect() such that the flags PROT_EXEC | PROT_WRITE cause an error to be returned and protection bits to not be set on the page
- Must also ensure that one cannot PROT_WRITE a page, then later remove that privilege and add PROT_EXEC

W^X

- W^X is a nice solution to several problems, but has it’s own drawbacks
  - Some architectures do not provide for an execute/no-execute bit on their pages (16-bit and 32-bit x86, ARM before ARMv6, several others)
  - Poses problems for Just-In-Time compilers and other dynamic code generation techniques

Address Space Layout Randomization

- ASLR refers to a technique wherein the address space of a program is randomized at runtime to prevent attackers from reliably jumping to known positions in code/data

- Usually, programs are compiled such that they always start at the same location in virtual memory, making linking functions and such very simple

- ASLR breaks this assumption and changes the location of the start of the program, as well as it's data, heap, stack, etc

ASLR

- ASLR is not a be-all, end-all solution
- There is still (usually) a limited window of entropy for the randomization
- Can be defeated by multiple copies of data and NOP-sleds