Computer Security
03. Program Hijacking

February 11, 2017
CS 419 © 2017 Paul Krzyzanowski

Bugs and mistakes

- Most penetrations are due to
  - Social engineering
  - Or bugs
- Attacked system may be further weakened because of poor access control rules
  - Violate principle of least privilege
- Cryptography won’t help us!
  - And cryptographic software can also be buggy … and often is

Assumptions can get you in trouble

- Unchecked assumptions can lead to vulnerabilities
- Attack:
  - Discover assumptions
  - Craft an exploit to render them invalid
- Two common assumptions
  - Buffer is large enough for the data
  - Integer overflow doesn’t exist

Robert Tappan Morris Jr.’s Internet Worm

Attacked VAX computers running BSD
1. Attempt to crack local passwords
   - Guess passwords via dictionary attack
   - 432 common passwords and combinations of account name and user name
2. Look for readable .rhost files — that may give you rsh access to another system
3. Do a buffer overflow exploit on fingerd via gets to load a small program
   - 99 lines of C
   - Program connects to sender and downloads the full worm
4. Use the DEBUG command of sendmail
   - Allowed remote command execution on a remote system

Then propagate the program onto any system you can log into
Buffer Overflows

Some high-profile buffer overflow attacks
- 2001: Code Red worm
  - Buffer overflow attack on Microsoft's IIS
- 2003: SQL Slammer
  - Buffer overflow attack on Microsoft's SQL Server
- 2003: X-Box attack
  - Buffer overflow attack bypasses license checking
- 2010: PS2 Independence exploit
  - Buffer overflow attack bypasses license checking

February 11, 2017
CS 419 © 2017 Paul Krzyzanowski

Still with us

A function known as getaddrinfo() that performs domain name lookups contains a buffer overflow bug that allows attackers to remotely execute malicious code. It can be exploited when vulnerable devices or apps make queries to attacker-controlled domain names or domain name servers or when they're exposed to man-in-the-middle attacks where the adversary has the ability to monitor and manipulate data passing between a vulnerable device and the open Internet. All versions of glibc after 2.9 are vulnerable.


What is a buffer overflow?

- Programming error that allows more data to be stored in an array than there is space
- Buffer = stack, heap, static data
- Overflow means...
  - Adjacent memory will be overwritten
  - Program data can be corrupted
  - New code can be injected
  - Unexpected transfer of control can be launched

The classic buffer overflow bug

gets() from OS X: © 1990,1992 The Regents of the University of California.

```c
char *buf;
if (!warned) {
    (void) write(STDERR_FILENO, w, sizeof(w) - 1);
    warned = 1;
}
for (s = buf; (c = getchar()) != 'n';)
    if (c == EOF)
        if (s == buf)
            return (NULL);
        else
            break;
    else
        *s++ = c;
*buf = 0;
return (buf);
```

Still with us

Extremely severe bug leaves dizzying number of software and devices vulnerable

Since 2008, vulnerable host-based and cloud systems have been open to remote hacking.

March 2013

© 2017 Paul Krzyzanowski
Buffer overflow examples

```c
void test(void) {
    char name[10];
    strcpy(name, "krzyzanowski");
}
```

That’s easy to spot!

Another example

```c
char configfile[256];
char *base = getenv("BASEDIR");
if (base != NULL)
    sprintf(configfile, "%s/config.txt", base);
else {
    fprintf(stderr, "BASEDIR not set\n");
}
```

How about this?

```c
char line[80];
while (gets(line) != NULL) {
    /* process a line of input */
}
```

You made unchecked assumptions on the maximum password length

```c
char passwd1[80], passwd2[80];
printf("Enter password: ");
gets(passwd1);
printf("Enter password again: ");
gets(passwd2);
if (strcmp(passwd1, passwd2) != 0) {
    fprintf(stderr, "passwords don't match\n");
    exit(1);
}
```

What’s the harm?

- Execute arbitrary code, such as starting a shell
  - Code injection, stack smashing
- Code runs with the privileges of the program
  - If the program is setuid root then you have root privileges
  - If the program is on a server, you can run code on that server
- Even if you cannot execution code...
  - You may crash the program
  - Denial of service attack
- Sometimes the crashed code can leave a core dump
  - You can access that and grab data the program had in memory

Buffer overflow attacks

To exploit a buffer overflow
- Identify overflow vulnerability in a program
  - Inspect source
  - Trace execution
  - Use fuzzing tools (more on that ...)
- Understand where the buffer is in memory and whether there is potential for corrupting surrounding data

Yet another example

```c
char line[80];
while (gets(line) != NULL) {
    /* process a line of input */
}
```

Classic

```c
char passwd1[80], passwd2[80];
printf("Enter password: ");
gets(passwd1);
printf("Enter password again: ");
gets(passwd2);
if (strcmp(passwd1, passwd2) != 0) {
    fprintf(stderr, "passwords don't match\n");
    exit(1);
}
```

You might not notice

```c
char passwd1[80], passwd2[80];
printf("Enter password: ");
gets(passwd1);
printf("Enter password again: ");
gets(passwd2);
if (strcmp(passwd1, passwd2) != 0) {
    fprintf(stderr, "passwords don't match\n");
    exit(1);
}
```
It's a bounds checking problem

- C and C++
  - Allow direct access to memory
  - Do not check array bounds
  - Functions often do not know array bounds
    - They just get passed a pointer to the start of the structure
- This is not a problem with strongly typed languages
  - Java, C#, Python, etc. check sizes of structures
- But C is in the top 3 of popular programming languages
  - Dominant for system programming & embedded systems

Programming at the machine level

- High level languages (even C) constrain you in
  - Access to variables (local vs. global)
  - Control flows in predictable ways
    - Loops, function entry/exit, exceptions
- At the machine code level
  - No restriction on where you can jump
    - Jump to the middle of a function... or to the middle of a C statement
    - Returns will go to whatever address is on the stack
    - Unused code can be executed (e.g., library functions you don't use)

Linux process memory map

```
0x08048000  High memory
            Loaded by exec

0x40000000  Stack

0x08048000  Program text
```

Stack overflows

- Overflows can occur when programs do not validate the length of
  data being written to a buffer
- This could be in your code or one of several "unsafe" libraries
  - strcpy(char *dest, const char *src);
  - strcat(char *dest, const char *src);
  - gets(char *s);
  - scanf(const char *format, ...)
  - Others...

The stack

- Overflows can occur when programs do not validate the length of
  data being written to a buffer
- This could be in your code or one of several "unsafe" libraries
  - strcpy(char *dest, const char *src);
  - strcat(char *dest, const char *src);
  - gets(char *s);
  - scanf(const char *format, ...)
  - Others...
Overflowing the buffer

What if s is >128 bytes?

Overwriting the return address

- If we overwrite the return address
  - We change what the program executes when it returns from the function

- “Benign” overflow
  - Overflow with garbage data
  - Chances are that the return address will be invalid
  - Program will die with a SEGFAULT
  - Availability attack

Address Uncertainty

- What if we’re not sure what the exact address is?
- NOP Slide = landing zone
  - Pre-pad the code with a bunch of NOP instructions
  - NOP moving a register to itself, adding 0, etc.
  - Set the return address on the stack to any address within the landing zone

Subverting control flow

Malicious overflow
- Fill the buffer with malicious code
- Overflow to overwrite saved %rbp
- Then overwrite saved %rsp (return address) with the address of the malicious code in the buffer

Off-by-one overflows

What if s is >128 bytes?
You overwrite %rbp and then the return address

What if s is >128 bytes?
You overwrite %rbp and then the return address
Safe functions aren't always safe

- Safe counterparts require a count
  - strcpy → strncpy
  - strcat → strncat
  - sprintf → snprintf
- But programmers can miscount!

```
char buf[512];
int i;
for (i=0; i<512; i++)
    buf[i] = stuff[i];
```

Off-by-one errors

- We can't overwrite the return address
- But we can overwrite one byte of the saved frame pointer
  - Least significant byte on Intel/ARM systems
  - Little-endian architecture

Off-by-one errors

- Depends on the compiler but...
  - Sometimes the compiler restores the old stack pointer from the saved frame pointer
  - Stack frame pointer will now point to the location of the buffer
- Stuff the buffer with
  - Local variables
  - "saved" %ebp
  - "saved" %rip (return address)
  - Malicious code, pointed to by "saved" %rip
  - When the function returns
    - It will return to the "saved" %rip, which points to malicious code in the buffer

Heap & text overflows

- Statically allocated variables & dynamically allocated memory (malloc) are not on the stack
- Heap data & static data do not contain return addresses
  - No ability to overwrite a return address
- Are we safe?

Linux process memory map

- Only local variables are on the stack
Memory overflow

The program

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char a[15];
char b[15];

int main(int argc, char **argv)
{
    strcpy(b, "abcdefghijklmnopqrstuvwxyz");
    printf("a=%s\n", a);
    printf("b=%s\n", b);
    exit(0);
}
```

The output

```
(a=qrstuvwxyz
b=abcdefghijklmnopqrstuvwxyz
```

Overwriting variables

We may be able to overflow a buffer and overwrite other variables in higher memory

For example
- Overwrite a file name
- Change a variable

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char afile[20];
char mybuf[15];

int main(int argc, char **argv)
{
    strncpy(afile, "/etc/secret.txt", 20);
    printf("planning to write to %s\n",
    afile);
    strcpy(mybuf, "/usr/paul/writehere.txt");
    printf("about to open afile=%s\n",
    afile);
    exit(0);
}
```

The exploit

The program takes the first two arguments from the command line
It copies `argv[1]` into a buffer with no bounds checking
It then calls the callback, passing it the message from the 2nd argument

- The exploit
  - Overflow the buffer
  - The overflow bytes will contain the address of the function you really want to call
  - They’re strings, so bytes with 0 in them will not work ... making this a more difficult attack

```c
/* callback */
int callback(const char* msg)
{
    printf("callback called: %s\n", msg);
}
```

```c
int main(int argc, char **argv)
{
    static char buffer[16];
    static int (*fp)(const char *msg);
    fp = (int(*)(const char *msg))(callback);
    strcpy(buffer, argv[1]);
    // call the callback
}
```

`printf` attacks
printf and its variants

- Standard C library functions for formatted output
  - printf: print to the standard output
  - fprintf: print formatted data to a FILE stream
  - printf, vprintf: print formatted data to a memory location
  - fprintf, vfprintf: print formatted data containing a pointer to argument list
  - printf, fprintf: print formatted data containing a pointer to argument list

- Usage
  printf(“The number %d in decimal is %x in hexadecimal”...
  printf(“%s”..., name);

- Programs often make mistakes with printf
  - Valid:
    printf(“%d”, 123);
  - Also accepted... but not right
    printf(“%d”, “hello, world!”);

- This works but exposes the risk that message may be changed maliciously

Dumping memory with printf

```
#include <stdio.h>
#include <string.h>
int show(char *buf)
{
    printf(buf); putchar(‘\n’); return 0;
}

int main(int argc, char **argv)
{
    char buf[256];
    if (argc == 2) {
        strcpy(buf, argv[1], 255);
        show(buf);
    }
}
```

```
$ ./tt hello
hello
```

printf does not know how many arguments it has. It deduces that from the format string.
If you don’t give it enough arguments, it keeps reading from the stack....

We can dump arbitrary memory by walking up the stack:
```
0xffffffff-0x0000000f-0x00003a9b-0x0002a21f
```

Bad usage of printf

```
#include <stdio.h>
#include <string.h>
int show(char *buf)
{
    printf(buf); putchar(‘\n’); return 0;
}

int main(int argc, char **argv)
{
    char buf[256];
    if (argc == 2) {
        strcpy(buf, argv[1], 255);
        show(buf);
    }
}
```

```
$ ./tt 100x, 160x, 160x, 160x
0x00000000-0x00003a9b-0x0002a21f-0x0003e82f
```

Print attacks

- What good is %n when it's just # of bytes written?
  - Can specify an arbitrary number of bytes in the format string
  - Will write the value 622404-622400 = 12480 = 0x12fa8
  - %-0x says to print a hex number and have it take 622,404 characters

- Note we have two controls here:
  1. Put more % controls in printf: this determines what we write in memory
     - Each % represents an argument — we don't care what they are but each one takes us to the next element on the stack... until we get to the one we want to overwrite. With %a
  2. Have printf output more data: this determines what we write in memory
     - E.G., with formats like %d%a
     - That defines what the value of %n will be that gets written to memory
Defending against hijacking attacks

Fix bugs

- Audit software
- Check for buffer lengths whenever adding to a buffer
- Search for unsafe functions
  - Use nm and grep to look for function names
- Use automated tools
  - Clockwork, CodeSonar, Coventry Parasoft, PolySpace, Checkmarx, PREfix, PVS-Studio, PCPCheck, Visual Studio
- Most compilers and/or linkers now warn against bad usage

```
Fix bugs: Fuzzing

- Technique for locating buffer overflow problems
- Enter long strings with well-defined patterns
  - E.g., "$$$$$$"
- If the app crashes
  - Search the core dump for "$" to find where it died
- Automated fuzzer tools help with this
- Or ... try to construct exploits using gdb
```

Don’t use C or C++

- Buffer overflows are mostly a C/C++ problem
- Most other languages feature
  - Run-time bounds checking
  - Parameter count checking
  - Disallow reading from or writing to arbitrary memory locations
- Hard to avoid in many cases

```
Specify & test code

- If it’s in the specs, it is more likely to be coded & tested
- Document acceptance criteria
  - “File names longer than 1024 bytes must be rejected”
  - “User names longer than 32 bytes must be rejected”
- Ensure consistent checks to the criteria across entire source
  - Example, you might #define limits in a header file but some files might use a mismatched number.
- Check results from printf
```

Dealing with buffer overflows: No Execute

Data Execution Protection (DEP)

- Disallow code execution in data areas - on the stack or heap
- Set MMU per-page execute permissions to no-execute
- Intel and AMD added this support in 2004

- Examples
  - Microsoft DEP (Data Execution Prevention) (since Windows XP SP2)
  - Linux PaX patches
  - OS X ≤10.5
No Execute – not a complete solution

No Execute Doesn’t solve all problems
– Some applications need an executable stack (LISP interpreters)
– Some applications need an executable heap (code loading/patching)
– Does not protect against heap & function pointer overflows
– Does not protect against printf problems

February 11, 2017

Return-to-libc

• Allows bypassing need for non-executable memory
  – We can still corrupt the stack...just not execute code from it
• No need for injected code
• Instead, reuse functionality within the exploited app
• Use a buffer overflow attack to create a fake frame on the stack
  – Transfer program execution to the start of a library function
    – libc = standard C library
    – Most common function to exploit: system
    – Runs the shell
    – New frame contains the parameters for the shell: the command
      – E.g., system("/bin/\"")

Return Oriented Programming (ROP)

• Overwrite return address with address of a library function
  – Does not have to be the start of the library routine
    – "borrowed chunks"
  – When the library gets to RET, that location is on the stack, under the attacker’s control
• Chain together sequences ending in RET
  – Build together "gadgets" for arbitrary computation
  – Buffer overflow contains a sequence of addresses that direct each successive RET instruction
• It is possible for an attacker to use ROP to execute arbitrary algorithms without injecting new code into an application
  – Removing dangerous functions, such as system, is ineffective
  – Make attacking easier: use a compiler that generates gadgets!

Dealing with buffer overflows: ASLR

Address Space Layout Randomization
– Dynamically loaded libraries used to be loaded in the same place each time, as was the stack & memory-mapped files
– Well-known locations make them branch targets in a buffer overflow attack
– Position stack and memory-mapped files to random locations
– Position libraries at random locations
  – Libraries must be compiled to produce position independent code
  – Implemented in
  – But...not all libraries (modules) can use ASLR
  – And it makes debugging difficult

Address Space Layout Randomization

• Entropy
  – How random is the placement of memory regions?
• Examples
  – Linux Exec Shield patch
    – 19 bits of stack entropy, 16-byte alignment > 500K positions
  – Windows 7
    – 8 bits of randomness for DLLs
      – Aligned to 64K page in a 16MB region: 256 choices
  – Windows 8
    – 24 bits for randomness on 64-bit processors: >16M choices

Dealing with buffer overflows: Canaries

Stack canaries
– Place a random integer before the return address on the stack
– Before a return, check that the integer is there and not overwritten: a buffer overflow attack will likely overwrite it

© 2017 Paul Krzyzanowski
ProPolice

IBM’s gcc patches
- Allocate arrays into higher memory in the stack
- Ensures that a buffer overflow attack will not clobber non-array variables

```
int a, b=999;
char s[5], t[7];
gets(s);
```

Stack canaries
- Again, not foolproof
- Heap-based attacks are still possible
- Performance impact
  - Need to generate a canary on entry to a function and check canary prior to a return
  - Minimal degradation ~8% for apache

Stack canaries

Function pointer protection
- Encrypt function pointers
  - Example: XOR with a random value
  - Any attempt to modify them will result in invalid addresses
- Degrades performance when function pointers are used

Safer libraries
- Compilers warn against unsafe strcpy or printf
- Ideally, fix your code!
- Sometimes you can’t recompile (e.g., you lost the source)
- Libsafe from Avaya Labs
  - Dynamically loaded library
  - Intercepts calls to unsafe functions
  - Validates that there is sufficient space in the current stack frame
    framepointer > strlen(src)

The end