Announcements

• Ethics form due today (11:59pm)!
• Homework #1 due Thursday
• Clarifying Homework 1:
  • Option 1: Part 1 and Part 2 use the same technology. In this case, the technology can be any technology.
  • Option 2: Part 1 and Part 2 use different technologies. In this case, Part 1 must consider a security-related technology. Part 2 can still be any technology.
Next Time: Basic Buffer Overflows

- Memory pointed to by `str` is copied onto stack...

  ```c
  void func(char *str) {
    char buf[126];
    strcpy(buf, str);
  }
  ```

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

  This will be interpreted as return address!

  - `strcpy` does NOT check whether the string at `*str` contains fewer than 126 characters.
What About This?

• Home-brewed range-checking string copy

void mycopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}
void main(int argc, char *argv[]) {
    if (argc==2)
        mycopy(argv[1]);
}

• 1-byte overflow: can’t change RET, but can change pointer to previous stack frame...
Frame Pointer Overflow

Little Endian architecture: overwritten byte is least significant byte

ATTACK CODE
Another Variant: Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then one can call F as (*P)(...)

![Diagram](https://via.placeholder.com/150)
Other Overflow Targets

• Format strings in C
  • We’ll walk through this one today

• Heap management structures used by malloc()
  • More details in section
  • Techniques have changed wildly over time

• These are all attacks you can look forward to in Lab #1 😊
Variable Arguments in C

• In C, can define a function with a variable number of arguments
  – Example: `void printf(const char* format, ...)`

• Examples of usage:

```c
printf("hello, world");
printf("length of \"%s\" = %d\n", str, str.length());
printf("unable to open file descriptor %d\n", fd);
```

Format specification encoded by special % characters

%\d, %i, %o, %u, %x, %X – integer argument
%s – string argument
%p – pointer argument (void *)
Several others
Format Strings in C

• Proper use of printf format string:

```c
int foo = 1234;
printf(“foo = %d in decimal, %X in hex”, foo, foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

• Sloppy use of printf format string:

```c
char buf[14] = “Hello, world!”;
printf(buf);
// should’ve used printf(“%s”, buf);
```

What happens if buffer contains format symbols starting with % ???

4/5/2021 CSE 484 - Spring 2021 9
Implementation of Variable Args

• Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time

```c
void printf(const char* format, ...) {
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
            case 'd':
                i = va_arg(ap, int); break;
            case 's':
                s = va_arg(ap, char*); break;
            case 'c':
                c = va_arg(ap, char); break;
            }
            /* etc. for each % specification */
        }
    }
    va_end(ap); /* restore any special stack manipulations */
}
```

This is simplified code, e.g., handles %d but not %10d
Closer Look at the Stack

printf(“Numbers: %d,%d”, 5, 6);

printf(“Numbers: %d,%d”);

... Saved FP ret/IP &str 5 6 Caller’s frame
<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Args</td>
</tr>
<tr>
<td>Addr 0xFF...F</td>
</tr>
</tbody>
</table>

Internal stack pointer starts here

... Saved FP ret/IP &str Caller’s frame
<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Args</td>
</tr>
<tr>
<td>Addr 0xFF...F</td>
</tr>
</tbody>
</table>

Internal stack pointer starts here
Format Strings in C

• Proper use of printf format string:

```c
int foo=1234;
printf(“foo = %d in decimal, %X in hex”, foo, foo);
```

This will print:
```
foo = 1234 in decimal, 4D2 in hex
```

• Sloppy use of printf format string:

```c
char buf[14] = “Hello, world!”;
printf(buf);
// should’ve used printf(“%s”, buf);
```
Format Strings in C

Proper use of printf format string:

```c
int foo=1234;
printf("foo = %d in decimal, %X in hex", foo, foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

Sloppy use of printf format string:

```c
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

This can be exploited to move printf’s internal stack pointer!
Viewing Memory

• \%x format symbol tells printf to output data on stack

```c
printf("Here is an int: \%x", i);
```

• What if printf does not have an argument?

```c
char buf[16]="Here is an int: \%x";
printf(buf);
```

• Or what about:

```c
char buf[16]="Here is a string: \%s";
printf(buf);
```
Viewing Memory

- \%x format symbol tells printf to output data on stack

```c
printf("Here is an int: \%x", i);
```

- What if printf does **not** have an argument?

```c
char buf[16]="Here is an int: \%x";
printf(buf);
```

  - Stack location pointed to by printf’s internal stack pointer will be interpreted as an int. *(What if crypto key, password, ...?)*

- Or what about:

```c
char buf[16]="Here is a string: \%s";
printf(buf);
```

  - Stack location pointed to by printf’s internal stack pointer will be interpreted as a pointer to a string
Writing Stack with Format Strings

• `%n` format symbol tells `printf` to write the number of characters that have been printed

```c
printf(“Overflow this!%n”, &myVar);
```

• Argument of `printf` is interpreted as destination address
• This writes 14 into `myVar` (“Overflow this!” has 14 characters)

• What if `printf` does not have an argument?

```c
char buf[16]=“Overflow this!%n”;
printf(buf);
```

• Stack location pointed to by `printf`’s internal stack pointer will be interpreted as address into which the number of characters will be written.
Summary of Printf Risks

- Printf takes a variable number of arguments
  - E.g., `printf("Here’s an int: %d", 10);`
- Assumptions about input can lead to trouble
  - E.g., `printf(buf)` when `buf=“Hello world”` versus when `buf=“Hello world %d”`
  - Can be used to advance printf’s internal stack pointer
  - Can read memory
    - E.g., `printf("%x")` will print in hex format whatever printf’s internal stack pointer is pointing to at the time
  - Can write memory
    - E.g., `printf("Hello%n");` will write “5” to the memory location specified by whatever printf’s internal SP is pointing to at the time
“Weird Machines”

• Way of thinking about exploits (the best way 😊)

• Treat each discrete side-effect as an ‘instruction’

• Synthesize a ‘program’ from these instructions

• This is now your exploit!
How Can We Attack This?

foo() {
    char buf[...];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf);  //vulnerable
}

If format string contains % then printf will expect to find arguments here...

What should the string returned by readUntrustedInput() contain??

Go to Canvas Quiz for today!

Different compilers / compiler options / architectures might vary
Using %n to Overwrite Return Address

**In foo()'s stack frame:**

Buffer with attacker-supplied input “string”

“... attackString%n”, attack code

&RET

SFP

RET

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.

(4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Why is “in” in quotes? C allows you to concisely specify the “width” to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf(“%5d%n”, 10) will print three spaces followed by the integer: “10”

That is, the %n will write 5, not 2.

Number of characters “in” attackString must be equal to … what?

When %n happens, make sure the location under printf’s stack pointer contains address of RET; %n will write the number of characters in printed so far into RET

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte. (4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

This portion contains enough % symbols to advance printf’s internal stack pointer
Recommended Reading

• It will be hard to do Lab 1 without:
  • Reading (see course schedule):
    • Smashing the Stack for Fun and Profit
    • Exploiting Format String Vulnerabilities
  • Attending section this week and next