CSE 484 / CSE M 584: Computer Security and Privacy

Autumn 2019

Tadayoshi (Yoshi) Kohno
yoshi@cs.Washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Franzi Roesner, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials...
Announcements

• Homework #1: Due Friday (10/4)
• Lab #1: Out, discussed in Quiz Section
• Day Before Thanksgiving: Alternate Video Lesson (e.g., use to support your final project)
• Final Project: Online, marked as draft but dates should be set
  – Linked off of Assignments page
  – 12-15 minute video on security-related topic of your choice
  – Note requirements, e.g., include references, discuss ethics/legal issues, length
David Aucsmith’s Lecture

• Questions?
• Observations?
• General Thoughts?
Example Topics

- Espionage vs warfare
- Cyber crime as a service
- Policies undecided (e.g., Apple vs FBI)
- Tempest
- Tor and Nation State Actors
- Supply Chain Security
- Future: Attacks on Trust
Last 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!

strncpy does NOT check whether the string at *str contains fewer than 126 characters.
Off-By-One Overflow

• Home-brewed range-checking string copy

```c
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]);
}
```

This will copy **513** characters into buffer. Oops!

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

Fake FP | Fake RET
---|---

buf | Saved FP | ret/IP | str
---|---|---|---

Local variables

Args

Addr 0xFF...F

ATTACK CODE
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:

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

• Risky 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 % ???
Background: 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 */
        }
    }
    ... /* etc. for each % specification */
}
va_end(ap); /* restore any special stack manipulations */
```
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
  ```

• Risky 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 % ??
Format Strings in C

Proper use of printf format string:

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

This will print:

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

Risky use of printf format string:

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

If the buffer contains format symbols starting with %, the location pointed to by printf’s internal stack pointer will be interpreted as an argument of printf.

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

What happens if buffer contains format symbols starting with %???
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 cryptographic 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.
foo() {
    char buf[...] = "attackString";
    printf(buf);  //vulnerable
}

What should “attackString” be??
To Do

• In addition to
  – HW 1
  – Lab 1

• Look at Final Project Description

• Think about format string vulnerabilities

• Think about how to defend against buffer overflow attacks