Required reading:
“StackGuard: Simple Stack Smash Protection for GCC”

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

Format string vulnerabilities
printf format strings

int i = 10;
printf("%d %p\n", i, &i);

- printf takes variable number of arguments
- printf pays no mind to where the stack frame “ends”
- It presumes that you called it with (at least) as many arguments as specified in the format string
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
Format string vulnerabilities

- `printf("%d %d %d %d ...");`
  - Prints a series of stack entries as integers

- `printf("%08x %08x %08x %08x ...");`
  - Same, but nicely formatted hex

- `printf("%s");`
  - Prints bytes pointed to by that stack entry

- `printf("100% done");`
  - Prints stack entry 4 bytes above saved %eip

- `printf("100% no way!");`
  - \textbf{WRTIES} the number 3 to address pointed to by stack entry
Format string prevalence

% of vulnerabilities that involve format string bugs

What’s wrong with this code?

```c
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }

    memcpy(buf, p, len);
}
```

Ok

```
int len = read_int_from_network();
char *p = read_string_from_network();
```

Negative

```
void *memcpy(void *dest, const void *src, size_t n);
```

*implicit cast to unsigned*

```c
typedef unsigned int size_t;
```
Integer overflow vulnerabilities
What's wrong with this code?

```c
void vulnerable()
{
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

**HUGE**

Takeaway: You have to know the semantics of your programming language to avoid these errors.
Integer overflow prevalence

% of vulnerabilities that involve integer overflows

What's wrong with this code?

Suppose that it has higher privilege than the user

```c
int main() {
    char buf[1024];
    ...
    int euid;
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    ln -s /usr/sensitive ~attacker/mystuff.txt
    printf("%s\n", buf);
    return 0;
}
```

```
uid euid
```

"Time of Check/Time of Use" Problem (TOCTOU)
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    seteuid(euid);  // Restore privileges
    printf(buf);
}
Memory safety attacks

- Buffer overflows
  - Can be used to read/write data on stack or heap
  - Can be used to inject code (ultimately root shell)

- Format string errors
  - Can be used to read/write stack data

- Integer overflow errors
  - Can be used to change the control flow of a program

- TOCTOU problem
  - Can be used to raise privileges
General defenses for memory-safety
Defensive coding practices

• Think defensive driving
  • Avoid depending on anyone else around you
  • If someone does something unexpected, you won’t crash (or worse)
  • It’s about *minimizing trust*

• Each module takes responsibility for checking the validity of all inputs sent to it
  • Even if you “know” your callers will never send a NULL pointer…
  • …Better to throw an exception (or even exit) than run malicious code

How to program defensively

- Code reviews, real or imagined
  - Organize your code so it is obviously correct
  - Re-write until it would be self-evident to a reviewer

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

- Remove the opportunity for programmer mistakes with better languages and libraries
  - Java performs automatic bounds checking
  - C++ provides a safe std::string class
Secure coding practices

char digit_to_char(int i) {
    char convert[] = “0123456789”;  
    return convert[i];
}

Think about all potential inputs, no matter how peculiar

char digit_to_char(int i) {
    char convert[] = “0123456789”;  
    if(i < 0 || i > 9) 
        return ‘?’;
    return convert[i];
}

Enforce rule compliance at runtime
**Rule:** Use safe string functions

- Traditional string library routines assume target buffers have sufficient length
  ```c
  char str[4];
  char buf[10] = "good";
  strcpy(str,"hello"); // overflows str
  strcat(buf," day to you"); // overflows buf
  ```

- Safe versions check the destination length
  ```c
  char str[4];
  char buf[10] = "good";
  strlcpy(str,"hello",sizeof(str)); // fails
  strlcat(buf," day to you",sizeof(buf)); // fails
  ```
Replacements

• … for string-oriented functions
  • `strcat` ⇒ `strlcat`
  • `strcpy` ⇒ `strlcpy`
  • `strncat` ⇒ `strlcat`
  • `strncpy` ⇒ `strlcpy`
  • `sprintf` ⇒ `snprintf`
  • `vsprintf` ⇒ `vsnprintf`
  • `gets` ⇒ `fgets`

• Microsoft versions different
  • `strcpy_s`, `strcat_s`, …

Note: None of these in and of themselves are “insecure.” They are just commonly misused.
(Better) **Rule**: Use safe string library

- Libraries designed to ensure strings used safely
  - **Safety first**, despite some performance loss

- Example: Very Secure FTP (**vsftpd**) **string library**

```
struct mystr; // impl hidden

void str_alloc_text(struct mystr* p_str, const char* p_src);
void str_append_str(struct mystr* p_str, const struct mystr* p_other);
int str_equal(const struct mystr* p_str1, const struct mystr* p_str2);
int str_contains_space(const struct mystr* p_str);
...
```

- Another example: **C++** `std::string` safe string library
**Rule: Understand pointer arithmetic**

```c
int x;
int *pi = &x;
char *pc = (char*) &x;

(pi + 1) == (pc + 1) ???
```

- `sizeof()` returns number of bytes, but pointer arithmetic multiplies by the `sizeof` the type.

```c
int buf[SIZE] = { ... };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
    *buf_ptr++ = getnext(); // will overflow
}
```

- So, use the right units

```c
while (!done() && buf_ptr < (buf + SIZE)) {
    *buf_ptr++ = getnext(); // stays in bounds
}
```
Defend dangling pointers

```c
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*));  // reuses p's space
*q = &x;
*p = 5;
**q = 3;  // crash (or worse)!
```

---

**IE's Role in the Google-China War**

Computer security companies are scrambling to cope with the fallout from the Internet Explorer (IE) flaw that led to cyberattacks on Google and its corporate and individual customers.

The zero-day attack that exploited IE is part of a lethal cocktail of malware that is keeping researchers very busy.

"We're discovering things on an up-to-the-minute basis, and we've seen about a dozen files dropped on infected PCs so far," Shmuel Alperovitch, vice president of research at McAfee Labs, told TechNewsWorld.

The attacks on Google, which appeared to originate in China, have sparked a feud between the Internet giant and the nation's government over censorship, and it could result in Google pulling away from its business dealings in the country.

**Pointing to the Flaw**

The vulnerability in IE is an invalid pointer reference, Microsoft said in security advisory 979352, which it issued on Thursday. Under certain conditions, the invalid pointer can be accessed after an object is deleted, the advisory states. In specially crafted attacks, like the ones launched against Google and its customers, IE can allow remote execution of code when the flaw is exploited.
Rule: Use NULL after free

```c
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; // defend against bad deref
int **q = malloc(sizeof(int*)); // reuses p's space
*q = &x;
*p = 5; // (good) crash
**q = 3;
```

![Stack and Heap diagram with pointers and values](image)
Manage memory properly

- Common approach in C: `goto chains` to avoid duplicated or missed code
- Like try/finally in languages like Java
- Confirm your logic!...

```c
int foo(int arg1, int arg2) {
    struct foo *pf1, *pf2;
    int retc = -1;
    pf1 = malloc(sizeof(struct foo));
    if (!isok(arg1)) goto DONE;
    ...
    pf2 = malloc(sizeof(struct foo));
    if (!isok(arg2)) goto FAIL_ARG2;
    ...
    retc = 0;
FAIL_ARG2:
    free(pf2); //fallthru
DONE:
    free(pf1);
    return retc;
}
```
Rule: Use a safe allocator

- ASLR challenges exploits by making the base address of libraries unpredictable

- **Challenge heap-based overflows** by making the addresses returned by `malloc` **unpredictable**
  - Can have some negative performance impact

- Example implementations:
  - **Windows Fault-Tolerant Heap**
  - **DieHard** (on which fault-tolerant heap is based)
    - [http://plasma.cs.umass.edu/emery/diehard.html](http://plasma.cs.umass.edu/emery/diehard.html)
Rule: Favor safe libraries

- **Libraries** encapsulate **well-thought-out design.**
  *Take advantage!*

- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard

- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient
This time

Continued with

Software Security

Writing & testing for

Secure Code

• Return oriented programming

• Format string & integer overflow vulnerabilities

• Defenses via good code & automated pen testing

(We’ll finish this up next time)
Defensive coding practices

• Think defensive driving
  • Avoid depending on anyone else around you
  • If someone does something unexpected, you won’t crash (or worse)
  • It’s about minimizing trust

• Each module takes responsibility for checking the validity of all inputs sent to it
  • Even if you “know” your callers will never send a NULL pointer…
  • …Better to throw an exception (or even exit) than run malicious code

Automated testing techniques

- **Code analysis**
  - Static: Many of the bugs we’ve shown could be easily detected (but we run into the Halting Problem)
  - Dynamic: Run in a VM and look for bad writes (valgrind)

- **Fuzz testing**
  - Generate **many random inputs**, see if the program fails
    - Totally random
    - Start with a valid input file and **mutate**
    - **Structure-driven** input generation: take into account the intended format of the input (e.g., “string int float string”)
  - Typically involves **many many** inputs (clusterfuzz.google.com)
Penetration testing

- Fuzz testing is a form of "penetration testing" (pen testing)

- Pen testing assesses security by actively trying to find exploitable vulnerabilities
  - Useful for both attackers and defenders

- Pen testing is useful at many different levels
  - Testing programs
  - Testing applications
  - Testing a network
  - Testing a server…
Fuzzing inputs

- **Mutation**
  - Take a [legal input and *mutate* it](https://example.com), using that as input
  - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query
    - Mutation might also be forced to adhere to grammar

- **Generational**
  - Generate input from scratch, e.g., from a [grammar](https://example.com)

- **Combinations**
  - Generate initial input, mutate\(^N\), generate new inputs, …
  - Generate mutations according to grammar
Kinds of fuzzing

• **Black box**
  • The tool knows nothing about the program or its input
  • **Easy to use** and get started, but will **explore only shallow states** unless it gets lucky

• **Grammar based**
  • The tool generates input informed by a grammar
  • **More work to use**, to produce the grammar, but **can go deeper** in the state space

• **White box**
  • The tool generates new inputs at least partially informed by the code of the program being fuzzed
  • Often **easy to use**, but **computationally expensive**
File-based fuzzing

- **Mutate** or **generate** inputs
- **Run the target program** with them
- **See what happens**
Examples: Radamsa and Blab

• **Radamsa** is a *mutation-based*, black box fuzzer
  • It mutates inputs that are given, passing them along

```bash
$ echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
$ echo … | radamsa --seed 12 -n 4 | bc -1
```

• **Blab** *generates* inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```bash
$ lab -e '((([wrstp][aeiouy]{1,2}){1,4} 32){5} 10' soty wypisi tisyro to patu
```

Network-based fuzzing

- **Act as 1/2 of a communicating pair**
  - Inputs could be produced by replaying previously recorded interaction, and altering it, or producing it from scratch (e.g., from a protocol grammar)
Network-based fuzzing

- Act as a “man in the middle”
  - mutating inputs exchanged between parties (perhaps informed by a grammar)