Dynamic Security Testing

Sicco Verwer
s.e.verwer@tudelft.nl
Today

• The world of software security

• How is it possible?
  • Integer overflows
  • Buffer overflows
  • Heartbleed
  • Stagefright

• How can it be prevented?
  • Fuzzing
  • Symbolic execution
  • Automated reversing

Many slides courtesy of Erik Poll (RU Nijmegen) and Dawn Song (Berkeley)
The world of software security
Who uses WhatsApp?

**You Should Update WhatsApp Right Away. Here's How to Do it**
TIME - 14 hours ago
A security vulnerability in Facebook-owned messaging app WhatsApp can reportedly allow hackers to gain access to your smartphone's ...

**WhatsApp: How to stay safe on social media**
BBC News - 17 hours ago
The phrase "WhatsApp targeted attack" is something no WhatsApp user wants to see in a headline. Add in "hackers were able to remotely ..."

**WhatsApp Was Hacked, Your Computer Was Exposed, and More News**
WIRED - 6 hours ago
The messaging platform WhatsApp is well known for its end-to-end encryption, but recent news calls its security into question. The NSO Group ...

**WhatsApp issues patch for spyware breach**
CNBC - 16 hours ago
Facebook's WhatsApp urged users to upgrade to the latest version of its popular messaging app after reporting that users might be vulnerable ...
Before hacking

- In 1950s, Joe Engressia showed the telephone network could be hacked by **phone phreaking**:
  - ie. by whistling at right frequencies

https://www.youtube.com/watch?v=vVZm7I1CTBs

- In 1970s, before founding Apple together with Steve Jobs, Steve Wozniak sold Blue Boxes for phone phreaking at university
Brief history of malware

- 1982:
  - Highschool student Rick Sorent wrote the Elk Cloner, the first computer virus that spread via floppy disks for Apple II
- 1988:
  - University student Robert Morris wrote the first internet worm, the Morris worm
    - Unintentionally, it crashed 10% of the internet.
    - First conviction under the 1986 US Computer Fraud and Abuse Act.
- late 1990/early 2000s, many more viruses and worms:
  - Email viruses: I Love You, Kournikova, ...
  - Worms: Slammer, CodeRed, MyDoom, Nimda, ...
Slammer worm (2003)

Pictures from *The Spread of the Sapphire/Slammer Worm*, by David Moore, Vem Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver
Slammer worm (2003)

Pictures from *The Spread of the Sapphire/Slammer Worm*, by David Moore, Vem Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver
Slammer worm (2003)

Pictures from *The Spread of the Sapphire/Slammer Worm*, by David Moore, Vem Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver
Slammer worm

- Exploited a buffer overflow in SQL Server (Microsoft)
- This bug was already patched six months earlier!
- A small piece of code that continuously generates random IP addresses and sends itself to those addresses
  - Only 376 bytes large
- Drastically slowed-down internet traffic
  - crashing numerous routers
  - causing a flood of routing table updates
Hacking turns professional

- Hacking not just fun, but **profitable**:
  - **stealing** user data (usernames & passwords, credit card no’s, ...)
  - sending **spam**, eg for **phishing**
  - interfering with internet transactions (eg internet **banking**)
  - new business models for making money:
    - **adware**, **scareware**, or **ransomware**
    - creating **botnets**, large collections of infected computers (bots), which can then be used for all of the above
  - **and for warfare, terrorism, espionage**,...
Stuxnet

- Advanced malware spread via USB sticks to stealthily target embedded software (SCADA systems in a Iranian nuclear facility using multiple (expensive) zero day vulnerabilities

Ralph Langer on stuxnet: [http://www.youtube.com/watch?v=CS01Hmjv1pQ](http://www.youtube.com/watch?v=CS01Hmjv1pQ)
Hacking today

- Several variants of Stuxnet found: Flame, Duqu, Gauss
  - All military grade malware, very hard to analyze due to advanced encryption

- Conficker worm found on French navy network

- Advanced attacks on infrastructure: Telvent attack
  - Closer to home: attacks on ports of Rotterdam and Antwerp

- Many instances of cyber espionage:
  - Nitro attack, Icefog, Putter Panda, PLATINUM, ...
A marketplace for vulnerabilities

- **Option 1: Bug bounty programs**
  - Google vulnerability reward program: 3k $
  - Mozilla Bug Bounty program: 500 $
  - Pwn2Own competition: 15k $

- **Option 2: Responsible disclosure**
  - ZDI, iDefense: 2k – 25k $

- **Option 3: Black market**
  - “some exploits”: 200K-250k $
  - A “real good exploit”: over 100k $

Source: Charlie Miller
(securityevaluators.com/files/papers/0daymarket.pdf)
Reporting vulnerabilities

http://www.us-cert.gov/ncas/alerts/
http://www.securitytracker.com/
http://www.securityfocus.com/vulnerabilities

- Such sites use different policies:
  - publishing all vulnerabilities
    - possibly only after some waiting period for responsible disclosure
  - only publishing those that are known to be exploited
  - only publishing those for which there is a patch
Keep your system up-to-date!

- Vulnerability announcements cause patches, but also hacks:

  Figures from *Before we knew it: An empirical study of zero-day attacks in the real world*, by Leyla Bilge and Tudor Dimitras
How bad is it for you?

- Someone can take full control of your PC
  - take screen shots
  - monitor keystrokes for login credentials
  - lock your system
  - use it for DDoS attacks
  - ...

- Nowadays, not only hackers can do so, little programming experience in required
  - Many sophisticated hacking tools exist with GUIs, all you need to know is how to use a mouse...
What causes the problem
What would you test?

- Testing increase i and decrease d, balance resets to 1000:

  - i(100) → 1100
  - i(1000) → 2000
  - d(100) → 900
  - d(1000) → 0
int balance = 1000;

void decrease(int amount)
{
    if (balance <= amount)
    {
        balance = balance - amount;
    }
    else
    {
        printf("Insufficient funds\n");
    }
}

void increase(int amount)
{
    balance = balance + amount;
}
Exercise: spot the bugs

```c
int balance = 1000;

void decrease(int amount)
{
    if (balance <= amount)
    {
        balance = balance - amount;
    }
    else
    {
        printf("Insufficient funds\n");
    }
}

void increase(int amount)
{
    balance = balance + amount;
}
```
Exercise: spot the bugs

```c
int balance = 1000;

void decrease(int amount)
{
    if (balance <= amount)
    {
        balance = balance - amount;
    }
    else
    {
        printf("Insufficient funds\n");
    }
}

void increase(int amount)
{
    balance = balance + amount;
}
```

should be `>=`
Exercise: spot the bugs

```c
int balance = 1000;

void decrease(int amount)
{
    if (balance <= amount)
    {
        balance = balance - amount;
    }
    else
    {
        printf("Insufficient funds\n");
    }
}

void increase(int amount)
{
    balance = balance + amount;
}
```

should be >=

what if amount is negative?
Exercise: spot the bugs

```c
int balance = 1000;

void decrease(int amount) {
    if (balance <= amount) {
        balance = balance - amount;
    } else {
        printf("Insufficient funds\n");
    }
}

void increase(int amount) {
    balance = balance + amount;
}
```

should be `>=`

what if amount is negative?

what if sum is too large for int?
Exercise: spot the bugs

```c
int balance = 1000;
void decrease(int amount) {
    if (balance <= amount) {
        balance = balance - amount;
    } else {
        printf("Insufficient funds\n");
    }
}
void increase(int amount) {
    balance = balance + amount;
}
```

- Should be `>=`
- What if amount is negative?
- What if sum is too large for `int`?

How to do this for thousands of lines of code….
Different implementation flaws

1. Logic error

2. Possible lack of input validation, problem when input is untrusted

3. Possible overflow, depends on underlying hardware
Common theme in flaws: untrusted input

• A very common source of security problems is assuming that input values will be `sensible’

*If an attacker can control the inputs, this assumption is false.*

• Many security flaws are caused untrusted inputs that are not checked aka validated, eg:
  • a numerical input can be negative
  • a numerical input might even not be numerical
  • an image file (eg a JPEG) may be malformed
  • a user might choose a 1 Mbyte long username or email address

• The only safe default is treating all input as untrusted!
Spot the bugs 2

#define MAX_BUF 256

void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);
    if (len < MAX_BUF)
        strcpy(buf,input);
}
#define MAX_BUF 256

void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);
    if (len < MAX_BUF)
        strcpy(buf,input);
}

max short = 32K
Spot the bugs 2

```c
#define MAX_BUF 256

void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);
    if (len < MAX_BUF)
        strcpy(buf,input);
}
```

max short = 32K

what if input is larger than 32K?
#define MAX_BUF 256

void BadCode (char* input) {
    short len;
    char buf[MAX_BUF];
    len = strlen(input);
    if (len < MAX_BUF) {
        strcpy(buf, input);
    }
}

max short = 32K

what if input is larger then 32K?

len will be negative

causin a buffer overflow…
What is a buffer overflow?

- Suppose in a C program we have an array of length 4:
  
  ```c
  char buffer[4];
  ```

- What happens if we execute the statement below?
  
  ```c
  buffer[4] = 'a';
  ```

- This is **UNDEFINED**! ANYTHING can happen!

- If the data written (ie. ‘a’) is user input that can be controlled by an attacker, this vulnerability can be exploited:

  
  *anything that the attacker wants can happen!*
The solution

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!

- Unfortunately, C and C++ have not adopted this solution for efficiency reasons
  - (Perl, Python, Java, C#, and even Visual Basic have)

- As a result, buffer overflows have been the no 1 security problem in software ever since
  - The first Internet worm, and all subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows
  - And they are still being exploited...
Points and memory

- Computer memory is a sequence of bytes, in hex notation:

  0x00   0x13
  1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0

- A pointer is a memory reference: $p^* = 0x05$

- In C you
  - copy pointer values to point to the same memory: $a = p$
  - dereference a pointer to access memory content: $b = *p$

- $a$ contains 0x05, $b$ contains 6
Pointers and memory

- Computer memory is a sequence of bytes, in hex notation

\[
\begin{array}{ccccccccccccccccccccccc}
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0
\end{array}
\]

- An array is a fixed pointer: char a[5]
- Pointing to a fixed length memory block
  - use arrays as pointers \( *p = a \)
  - offset the pointer value \( b = p + 1 \)
  - dereference array values using brackets \( c = a[1] \)
  - but also works for pointers \( d = b[1] \)

- p contains 0x09, b contains 0x10, c contains 1, d contains 2
The Stack

- When calling functions, memory is allocated to hold local variables, this memory is called the stack.

- The stack grows when calling functions.
- The stack decreases when returning.

- Every function call gets assigned its own stack frame, simply a block of memory similar to an array.
Challenge the future

StackFrame!

user%stack%
shared%libraries%
run%time%heap
static%data
segment
unused
text%segment
(program)

-0xC0000000
-0x40000000
-0x08048000
-0x00000000

arguments
return%address
stack%frame%pointer
exception%handlers
local%variables
callee%saved%registers

To!previous!stack!frame!pointer
To!the!point!at!which!
this%function%was%called
Stack Frame

A quick example to illustrate multiple stack frames

Example and slides from Dawn Song
What are buffer overflows?

**parse.c**

```c
void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i]!='\0' && in[i]!='\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}
```

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0; }
```

/** main to load a file and run parse */

**file** (input file)

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

parse’s frame

```
0xbffff760 0x0804a008
0xbffff75c 0x080485a2
0xbffff758 0xbffff778
0xbffff74c 0xbffff6c4
0xbffff748 0x00000001
0xbffff744 0xbffff778
0xbffff740 0xbffff6c4
0xbffff73c 0xbffff6b8
0xbffff6c0 0x20544547
0xbffff6b4 0xbffff740
0xbffff6b0 0xbffff6c4
0xbffff6ac 0x080485a2
0xbffff6a8 0xbffff758
0xbffff69c 0x00000000
```

(fp) return address stack frame ptr

(url) header_ok buf[4] buf[3,2,1,0] cmd[128,127,126,125]...

(cmd[7,6,5,4] cmd[3,2,1,0])

(out) in return address stack frame ptr

(copy_lower’s frame)
What are buffer overflows?

```c
40
Challenge the future

What are buffer overflows?!
```

```c
void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i]!='\0' && in[i]!='\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}
```

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0; }
```
What are buffer overflows?

### parse.c

```c
1: void copy_lower (char* in, char* out) {  
2:   int i = 0;  
3:   while (in[i] != '\0' && in[i] != '\n') {  
4:     out[i] = tolower(in[i]);  
5:     i++;  
6:   }  
7:   buf[i] = '\0';  
8: }
9: int parse(FILE *fp) {  
10:   char buf[5], *url, cmd[128];  
11:   fread(cmd, 1, 256, fp);  
12:   int header_ok = 0;  
13:   url = cmd + 4;  
14:   copy_lower(url, buf);  
15:   printf("Location is %s\n", buf);  
16:   return 0;  
17: }
21: /** main to load a file and run parse */
```

### file (input file)

```c
GETAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff760</td>
<td>0x0804a008</td>
<td>return address</td>
</tr>
<tr>
<td>0xbffff75c</td>
<td>0x080485a2</td>
<td>stack frame ptr</td>
</tr>
<tr>
<td>0xbffff758</td>
<td></td>
<td>url</td>
</tr>
<tr>
<td>0xbffff74c</td>
<td>0xbffff6c4</td>
<td>header_ok</td>
</tr>
<tr>
<td>0xbffff748</td>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>0xbffff744</td>
<td>0xbfe020dc</td>
<td></td>
</tr>
<tr>
<td>0xbffff740</td>
<td>0xbf022261</td>
<td></td>
</tr>
<tr>
<td>0xbffff73c</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0xbffff6c4</td>
<td>0x41414141</td>
<td></td>
</tr>
<tr>
<td>0xbffff6c0</td>
<td>0x20544547</td>
<td></td>
</tr>
<tr>
<td>0xbffff6b4</td>
<td>0xbffff740</td>
<td></td>
</tr>
<tr>
<td>0xbffff6b0</td>
<td>0xbffff6c4</td>
<td></td>
</tr>
<tr>
<td>0xbffff6ac</td>
<td>0x080485a2</td>
<td></td>
</tr>
<tr>
<td>0xbffff6a8</td>
<td>0xbffff758</td>
<td></td>
</tr>
<tr>
<td>0xbffff69c</td>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>

(Unallocated)
What are buffer overflows?

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

void copy_lower(char* in, char* out) {
    while (in[i] != '\0' && in[i] != '\n') {
        out[i] = tolower(in[i]);
    }
    buf[i] = '\0';
}

int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    fread(cmd, 1, 256, fp);
    int header_ok = 0;
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}

int main() {
    FILE *fp = fopen("input", "r");
    if (fp == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    parse(fp);
    return 0;
}
```

(Unallocated)
What are buffer overflows?

parse.c

```c
1: void copy_lower (char* in, char* out) {
  2:   int i = 0;
  3:   while (in[i]!=’\0’ & in[i]!=’\n’) {
  4:     out[i] = tolower(in[i]);
  5:     i++;
  6:   }
  7:   buf[i] = ‘\0’;
  8: }

9: int parse(FILE *fp) {
  10:   char buf[5], *url, cmd[128];
  11:   fread(cmd, 1, 256, fp);
  12:   int header_ok = 0;
  .
  19:   url = cmd + 4;
  20:   copy_lower(url, buf);
  21:   printf("Location is %s\n", buf);
  22:   return 0; }

23: /** main to load a file and run parse */
```
What are buffer overflows?

**parse.c**

```c
1: #is_int parse(FILE *fp) { 
2:     int i = 0; 
3:     while (in[i]!='\0' && in[i]!='\n') { 
4:         out[i] = tolower(in[i]); 
5:         i++; 
6:     } 
7:     buf[i] = '\0'; 
8: } 
9: #is int copy_lower (char* in, char* out) { 
10:     int i = 0; 
11:     while (in[i]!='\0' && in[i]!='\n') { 
12:         out[i] = tolower(in[i]); 
13:         i++; 
14:     } 
15:     buf[i] = '\0'; 
16: } 
17: int main() { 
18:     char buf[5], *url, cmd[128]; 
19:     url = cmd + 4; 
20:     copy_lower(url, buf); 
21:     printf("Location is %s\n", buf); 
22:     return 0; } 
```
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!='0' && in[i]!='\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }

9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    :.
19:    url = cmd + 4;
20:    copy_lower(url, buf);
21:    printf("Location is %s\n", buf);
22:    return 0; }
23: /* main to load a file and run parse */
```

(file (input file)

GET AAAAAA

parse.c

0xbffff760 0x0804a008
0xbffff75c 0x080485a2
0xbffff758 0xbffff778
0xbffff74c 0xbffff6c4
0xbffff748 0xbffff2061
0xbffff744 0x61616161
0xbffff740 0xbffff778
0xbffff73c 0x00000000
0xbffff6c4 0x41414141
0xbffff6c0 0x20544547
0xbffff6b4 0xbffff740
0xbffff6b0 0xbffff6c4
0xbffff6ac 0x080485a2
0xbffff6a8 0xbffff758
0xbffff69c 0x00000004

(Unallocated)
What are buffer overflows?

```
parse.c

1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!={'\0' && in[i]!=='\n'}) {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }

9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    url = cmd + 4;
14:    copy_lower(url, buf);
15:    printf("Location is %s\n", buf);
16:    return 0; }

23: /** main to load a file and run parse */

```

Uh oh….
What are buffer overflows?

**parse.c**

```c
1: void copy_lower (char* in, char* out) {
2:   int i = 0;
3:   while (in[i]!='\0' && in[i] != '\n') {
4:     out[i] = tolower(in[i]);
5:     i++;
6:   }
7:   buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:  char buf[5], *url, cmd[128];
11:  fread(cmd, 1, 256, fp);
12:  int header_ok = 0;
13:  ...
14:  url = cmd + 4;
15:  ...
16:  printf("Location is %s\n", buf);
17:  return 0; }
23: /** main to load a file and run parse */
```

**file** (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Uh oh....
What are buffer overflows?

`parse.c`

```c
void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i] != '\' \&\& in[i] != '
') {
        out[i] = tolower(in[i]);
        i++;
    }
    buf[i] = '\0';
}
```

```c
int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    int header_ok = 0;
    while (fread(cmd, 1, 256, fp));
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0;
}
```

Uh oh….!
What are buffer overflows?

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!=\0' & & in[i]!=\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    url = cmd + 4;
14:    copy_lower(url, buf);
15:    printf("Location is %s\n", buf);
16:    return 0; }
23: /** main to load a file and run parse **/
```

Uh oh....
Challenge the future

What are buffer overflows?

**parse.c**

```
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!='\0' && in[i]!='\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...;
14:    url = cmd + 4;
15:    copy_lower(url, buf);
16:    printf("Location is %s\n", buf);
17:    return 0; }
18:/** main to load a file and run parse */
```

**file** *(input file)*

GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Uh oh....
What are buffer overflows?

And when you try to return from parse... ... SEGFAULT, since 0x61616161 is not a valid location to return to.
Overwriting memory

• Overwriting the return address and thereby causing SEGFAULTS causes programs to crash.

• But this is not the main problem, by overwriting the memory now contains the input file name (AAAAA...A).

• In other words, the user input is in control of what gets written in the program's memory!

• Suppose we replace this with actual commands, called shellcode...
Basic Stack Exploit

```c
1: void copy_lower (char* in, char* out) { 
2:     int i = 0; 
3:     while (in[i]!='\0' && in[i]!='\n') { 
4:         out[i] = tolower(in[i]); 
5:         i++; 
6:     } 
7:     buf[i] = '\0'; 
8: } 
9: int parse(FILE *fp) { 
10:     char buf[5], *url, cmd[128]; 
11:     fread(cmd, 1, 256, fp); 
12:     int header_ok = 0; 
: 
13:     url = cmd + 4; 
14:     copy_lower(url, buf); 
15:     printf("Location is %s\n", buf); 
16:     return 0; } 
23: /** main to load a file and run parse */ 
```

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\x7f\xff \xffAAAA\x60\x1f\x5e \x89\x76\x08\x31\xc0\x88\x46\x46\x0c\x89\x4f \x88\x4f\x8d\x8e\x8b\x8d\x56\x0c\x8c\x80\x31\xdb \x89\x8d\x40\x8c\x80\xe8\x9c\x8d\x56\x0c\x8d\x8e\x8b\x8d\x56\x0c\x8c\x80\x31\xdb
```

(Unallocated)
Basic Stack Exploit

```
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!='\0' && in[i]!='\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:     char buf[5], *url, cmd[128];
11:     fread(cmd, 1, 256, fp);
12:     int header_ok = 0;
13:     url = cmd + 4;
14:     copy_lower(url, buf);
15:     printf("Location is %s\n", buf);
16:     return 0; }
23: /** main to load a file and run parse */
```
Basic Stack Exploit

**parse.c**

```c
1: void copy_lower (char* in, char* out) {
  2:   int i = 0;
  3:   while (in[i] != '\0' && in[i] != '\n') {
  4:     out[i] = tolower(in[i]);
  5:     i++;
  6:   }
  7:   buf[i] = '\0';
  8: }
  9: int parse(FILE *fp) {
 10:   char buf[5], *url, cmd[128];
 11:   fread(cmd, 1, 256, fp);
 12:   int header_ok = 0;
 13:   
 14:   url = cmd + 4;
 15:   copy_lower(url, buf);
 16:   printf("Location is %s\n", buf);
 17:   return 0;
 18: }
```
Basic Stack Exploit

parse.c

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i]!='\0' && in[i]!='\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...
14:    url = cmd + 4;
15:    copy_lower(url, buf);
16:    printf("Location is %s
", buf);
17:    return 0; }
20: ```

(file) (input file)

```
GET   AAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xff\x9f\xbx1f\x9e
\x89\x76\x08\x31\xcd\x88\x46\x0c\x80\x0b
\x89\x43\x8d\x4e\x08\x8d\x56\x3c\xcd\x80\x31\xdb
\x89\x4d\x08\xcd\x80\x8d\xdc\xff\xf7/bin/sh
```

OVERWRITE POINT!

```
0xbffff760  0x0804a008  fp
0xbffff75c  0x08fff764  return address
0xbffff758  0x61616161  stack frame ptr
0xbffff74c  0x61616161  url
0xbffff748  0x61616161  header_ok
0xbffff744  0x61616161  buf[4]
0xbffff740  0x61616161  cmd[128,127,126,125]
0xbffff73c  0x00000000  cmd[25,26,27,28]
0xbffff738  0x61616161  cmd[7,6,5,4]
0xbffff734  0x61616161  cmd[3,2,1,0]
0xbffff730  0x20544547

0xbffff6b4  0xbffff740  out
0xbffff6b0  0xbffff6c4  in
0xbffff6ac  0x080485a2  return address
0xbffff6a8  0xbffff758  stack frame ptr
0xbffff69c  0x00000019  i
```

(Unallocated)
Basic Stack Exploit

**parse.c**

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i] != ‘\0’ && in[i] != ‘\n’) {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = ‘\0’;
8: }
9: int parse(FILE *fp) {
10:     char buf[5], *url, cmd[128];
11:     fread(cmd, 1, 256, fp);
12:     int header_ok = 0;
13:     ...
14:     url = cmd + 4;
15:     copy_lower(url, buf);
16:     printf("Location is %s\n", buf);
17:     return 0; }
23: /** main to load a file and run parse */
```

**file** (input file)

```
GET   AAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xffAAA\web\x1f\xe5e
\x89\x76\x08\x31\xc0\x88\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xc0\x80\x31\xdb
\x89\x4d\x04\x8d\x80\xe8\xd0\xff\xf7\xff/bin/sh
```
**Basic Stack Exploit**

```c
void copy_lower (char* in, char* out) {
    int i = 0;
    while (in[i] != '\0' && in[i] != '\n') {
        out[i] = tolower(in[i]);
        i++;
    }
    out[i] = '\0';
}

int parse(FILE *fp) {
    char buf[5], *url, cmd[128];
    int header_ok = 0;
    ...
    url = cmd + 4;
    copy_lower(url, buf);
    printf("Location is %s\n", buf);
    return 0; }
}
```

`parse.c`

**file** (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x64\xf7\xff
\xff\xff\xff\xff\xfe\x05\x0e\x08\x31\x88\x46\x0c\x0b\x8d\x56\x0c\x80\x31\x8d
\x89\x8d\x40\xc0\x80\x8e\xc0\xff\xff\xff\xff\xff\xff\xff\xff\xff/bin/sh
```

**shellcode**

```
| 0xbffff764 |
| 0xbffff760 |
| 0xbffff75c |
| 0xbffff758 |
| 0xbffff74c |
| 0xbffff748 |
| 0xbffff744 |
| 0xbffff740 |
| 0xbffff73c |
| 0xbffff730 |
| 0xbffff72d |
| 0xbffff72c |
| 0xbffff72a |
| 0xbffff728 |
| 0xbffff724 |
| 0xbffff720 |
| 0xbffff71c |
| 0xbffff718 |
| 0xbffff714 |
| 0xbffff710 |
| 0xbffff70c |
| 0xbffff708 |
| 0xbffff704 |
| 0xbffff700 |
| 0xbffff6f8 |
| 0xbffff6f4 |
| 0xbffff6f0 |
| 0xbffff6e6 |
| 0xbffff6e2 |
| 0xbffff6e0 |
| 0xbffff6d6 |
| 0xbffff6d2 |
| 0xbffff6d0 |
| 0xbffff6c6 |
| 0xbffff6c2 |
| 0xbffff6c0 |
| 0xbffff6b6 |
| 0xbffff6b2 |
| 0xbffff6b0 |
| 0xbffff6a6 |
| 0xbffff6a2 |
| 0xbffff6a0 |
| 0xbffff69c |
| 0xbffff698 |
| 0xbffff694 |
| 0xbffff690 |
```

**ACTIVATE POINT!**

`fp`

`return address`

`stack frame ptr`

`url`

`header_ok`

`buf[3,2,1,0]`

`cmd[128,127,126,125]`

`cmd[25,26,27,28]`

`cmd[7,6,5,4]`

`cmd[3,2,1,0]`

`out`

`in`

`return address`

`stack frame ptr`

`n`

Basic Stack Exploit

**parse.c**

```c
1: void copy_lower (char* in, char* out) {
2:     int i = 0;
3:     while (in[i] != '\0' && in[i] != '\n') {
4:         out[i] = tolower(in[i]);
5:         i++;
6:     }
7:     buf[i] = '\0';
8: }
9: int parse(FILE *fp) {
10:    char buf[5], *url, cmd[128];
11:    fread(cmd, 1, 256, fp);
12:    int header_ok = 0;
13:    ...
19:    url = cmd + 4;
20:    copy_lower(url, buf);
21:    printf("Location is %s\n", buf);
22:    return 0; }
```

**file** (input file)

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

**user gets shell!**
How to attack this code?

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```
How to attack this code?

char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}

third argument expects an unsigned int
How to attack this code?

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

len is implicitly cast from int to unsigned int!
How to attack this code?

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

provide a negative value for len

if statement is happy

but the cast makes a negative len a very large int! causing a buffer overflow…
Spot the bugs 3

#ifdef UNICODE
#define _snprintf_snwprintf
#define TCHAR wchar_t
#else
#define _snprintf_snprintf
#define TCHAR char
#endif

TCHAR buff[MAX_SIZE];
_snprintf(buf, sizeof(buf), "%s\n", input);
Spot the bugs 3

#define _sntprintf _snwprintf
#define TCHAR wchar_t

#define _sntprintf _snprintf
#define TCHAR char

#elif defined_UNICODE

TCHAR buff[MAX_SIZE];
_sntprintf(buff, sizeof(buff), "%s\n", input);
Spot the bugs 3

```c
#ifdef UNICODE
#define _snprintf _snwprintf
#define TCHAR wchar_t
#else
#define _snprintf _snprintf
#define TCHAR char
#endif

TCHAR buff[MAX_SIZE];
_snprintf(buff, sizeof(buff), "%s\n", input);
```

_\texttt{snprintf}'s 2\textsuperscript{nd} argument is \# of chars in buffer, not \# of bytes

The CodeRed worm exploited such an mismatch, where code written under the assumption that 1 char was 1 byte allowed buffer overflows after the move from ASCI to Unicode.

From presentation by John Pincus
Stack/heap exploits

- Overwrite memory to contain your own code, or some library/shellcode of interest

- Not easy:
  - Have to determine return address (include NOP commands)
  - Overflow should not crash program before function exits
  - Shellcode may not contain ‘\0’ causing string to end

- But very powerful:
  - Any code can be executed, eg. granting system access

- Bugs that make them possible are hard to spot!
  - Avoid making input assumptions, be paranoid!
Not unique to C/C++

- Memory safe languages such as Java can trigger buffer overflows, eg. due to graphic libraries relying on fast native code:

**CVE reference:** CVE-2007-0243, **Release Date:** 2007-01-17

**Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability**

**Critical:** Highly critical, **Impact:** System access, **Where:** From remote

**Description:**

A vulnerability has been reported in Sun Java Runtime Environment (JRE), which can be exploited by malicious people to compromise a vulnerable system. The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a heap-based buffer overflow via a specially crafted GIF image with an image width of 0. Successful exploitation allows execution of arbitrary code.
What would you test?

- Testing a response system:

```
hello 5  =>  hello
hi 2   =>  hi
...
```
Spot the bug...

/* Read type and payload length first */

hbtype = *p++;
n2s(p, payload);
pl = p;
...
unsigned char *buffer, *bp; int r;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
Missing bound check

/* Read type and payload length first */

hbtype = *p++;
n2s(p, payload);
pl = p;
...

unsigned char *buffer, *bp; int r;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...

*bp++ = TLS1_HC_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

put payload length in payload, pl is pointer to actual payload
Missing bound check

/* Read type and payload length first */

hbtype = *p++;
n2s(p, payload);
pl = p;
...

unsigned char *buffer, *bp; int r;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...

*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

put payload length in payload, pl is pointer to actual payload
allocate up to 65535+1+2+16 of memory
Missing bound check

/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
...
unsigned char *buffer, *bp; int r;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
Missing bound check

/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
...
unsigned char *buffer, *bp; int r;
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

pl and payload are input and should not be trusted!

put payload length in payload, pl is pointer to actual payload

allocate up to 65535+1+2+16 of memory

copy memory from pl pointer to bp pointer of length payload
April 7, 2014: discovered that 2/3d of all web servers in world leak passwords. Programming oversight due to insufficient testing. #heartbleed #openssl
Who is to blame?

C/C++? – speed can be important

The OpenSSL developers? – a small group of volunteers with little funds

Vague specification? – should specifications cover all security bugs?

Functionality over security? – who uses heartbeat?

OpenSSL users? – billion dollar companies using free software without security audits...

April 7, 2014: discovered that 2/3d of all web servers in world leak passwords. Programming oversight due to insufficient testing. #heartbleed #openssl
Another example, July 2015
@ -330,6 +330,10 @

status_t SampleTable::setTimeToSampleParams
...

mTimeToSampleCount = U32_AT(&header[4]);
uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
if (allocSize > SIZE_MAX) {
    return ERROR_OUT_OF_RANGE;
}
mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
size_t size = sizeof(uint32_t) * mTimeToSampleCount * 2;
...

Spot the bug...
Spot the bug...

in C, multiplying two 32-bit ints, gives a 32-bit int

```c
@@ -330,6 +330,10 @@ status_t SampleTable::setTimeToSampleParams

    mTimeToSampleCount = U32_AT(&header[4]);
    uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
    if (allocSize > SIZE_MAX) {
        return ERROR_OUT_OF_RANGE;
    }
    mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
    size_t size = sizeof(uint32_t) * mTimeToSampleCount * 2;
```
Spot the bug...

in C, multiplying two 32-bit ints, gives a 32-bit int

@@ -330,6 +330,10 @@ status_t SampleTable::setTimeToSampleParams

mTimeToSampleCount = U32_AT(&header[4]);
uint64_t allocSize = mTimeToSampleCount * 2 * sizeof(uint32_t);
if (allocSize > SIZE_MAX) {
    return ERROR_OUT_OF_RANGE;
}
mTimeToSample = new uint32_t[mTimeToSampleCount * 2];
size = sizeof(uint32_t) * mTimeToSampleCount * 2;

check for security problem does not work since upper 32-bits are not checked!
How bad is it? Worst exploit: MMS

- Media is AUTOMATICALLY processed ON MMS RECEIPT.
  - BEFORE creating a notification!
    - Actually, while creating the notification

- Exploiting a vulnerability in Stagefright via MMS could allow SILENT, REMOTE, PRIVILEGED code execution.

- The attacker's payload simply needs to prevent the notification.

- Who has your phone number?
  - *When was the last time you updated your phone?*
Another example, july 2015

Who is to blame?

C/C++? – speed can be important..

The developer that wrote this code?

The compiler for not raising a warning?

Why are these errors even possible....
Wana Decrypt0r 2.0

Ooops, your files have been encrypted!

What Happened to My Computer?
Your important files are encrypted. Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?
Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don’t pay in 7 days, you won’t be able to recover your files forever.
We will have free events for users who are so poor that they couldn’t pay in 6 months.

How Do I Pay?
Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am GMT from Monday to Friday.

Send $300 worth of bitcoin to this address:

12t9YDPgwueZ9NyMgw519p7AAbisjr6SMw

Check Payment

Decrypt
Spot the bug

```c
int __stdcall SrvOs2FeaListSizeToNt(_DWORD *a1) {
    _WORD *v1; unsigned int v2; unsigned int v3; int v4; int v6;
    v1 = a1; v6 = 0;
    v2 = (unsigned int)a1 + *a1;
    v3 = (unsigned int)(a1 + 1);
    if ( (unsigned int)(a1 + 1) < v2 ) {
        while ( v3 + 4 < v2 ) {
            v4 = *(_WORD *)(v3 + 2) + *(_BYTE *)(v3 + 1);
            if ( v4 + v3 + 4 + 1 > v2 ) break;
            if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFFC, &v6) < 0 ) return 0;
            v3 += v4 + 5;
            if ( v3 >= v2 ) return v6;
            v1 = a1;
        }
    }
    *v1 = (_WORD)(v3 - v1);
} return v6; }
```
int __stdcall SrvOs2FeaListSizeToNt(_DWORD *a1) {
    _WORD *v1; unsigned int v2; unsigned int v3; int v4; int v6;
    v1 = a1; v6 = 0;
    v2 = (unsigned int)a1 + *a1;
    v3 = (unsigned int)(a1 + 1);
    if ( (unsigned int)(a1 + 1) < v2 ) {
        while ( v3 + 4 < v2 ) {
            v4 = *(__WORD *)(v3 + 2) + *(__BYTE *)(v3 + 1);
            if ( v4 + v3 + 4 + 1 > v2 ) break;
            if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFC, &v6) < 0 ) return 0;
            v3 += v4 + 5;
        }
        *v1 = (_WORD)(v3 - v1);
    } return v6; }
int _stdcall SrvOs2FeaListSizeToNt(_DWORD *a1) {
  _WORD *v1; unsigned int v2; unsigned int v3; int v4; int v6;

  v1 = a1; v6 = 0;
  v2 = (unsigned int) a1 + *a1;
  v3 = (unsigned int) (a1 + 1);

  if ( (unsigned int) (a1 + 1) < v2 ) {
    while ( v3 + 4 < v2 ) {
      v4 = *(_WORD *)(v3 + 2) + *(_BYTE *)(v3 + 1);
      if ( v4 + v3 + 4 + 1 > v2 ) break;
      if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFC, &v6) < 0 ) return 0; 
      v3 += v4 + 5;
    }
    v1 = a1;
    while ( v3 < v2 ) {
      v4 = *(_WORD *)(v3 + 2) + *(_BYTE *)(v3 + 1);
      if ( v4 + v3 + 4 + 1 > v2 ) break;
      if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFC, &v6) < 0 ) return 0; 
      v3 += v4 + 5;
    }
  }
  *v1 = (_WORD) (v3 - v1);
  return v6; }

But *v1 is
SMB_FEA_LIST->SizeOfListInBytes
which is a DWORD (32 bits)

puts a WORD (16 bits) into what is at address v1
Spot the bug

```c
int __stdcall SrvOs2FeaListSizeToNt(_DWORD *a1) {

    _WORD *v1; unsigned int v2; unsigned int v3; int v4; int v6;

    v1 = a1; v6 = 0;
    v2 = (unsigned int)a1 + *a1;
    v3 = (unsigned int)(a1 + 1);

    if ( (unsigned int)(a1 + 1) < v2 ) {
        while ( v3 + 4 < v2 ) {
            v4 = *(_WORD *)(v3 + 2) + *(_BYTE *)(v3 + 1);
            if ( v4 + v3 + 4 + 1 > v2 ) break;
            if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFC, &v6) < 0 ) return 0;
            v3 += v4 + 5;
            if ( v3 >= v2 ) return v6;
        }
    }

    *v1 = (_WORD)(v3 - v1);

    return v6;
}
```

But *v1 is SMB_FEA_LIST->SizeOfListInBytes

So if *v1 contains a large value 0x10000 and the assignment puts 0x0FFFFF (MAX WORD) into it the result is 0x1FFFFF, instead of the intended 0x0FFFFF.
Spot the bug

```c
int __stdcall SrvOs2FeaListSizeToNt(_DWORD *a1) {
    _WORD *v1; unsigned int v2; unsigned int v3; int v4; int v6;
    v1 = a1; v6 = 0;
    v2 = (unsigned int)a1 + *a1;
    v3 = (unsigned int)(a1 + 1);
    if ( (unsigned int)(a1 + 1) < v2 ) {
        while ( v3 + 4 < v2 ) {
            v4 = *(_WORD *)(v3 + 2) + *(_BYTE *)(v3 + 1);
            if ( v4 + v3 + 4 + 1 > v2 ) break;
            if ( RtlSizeTAdd(v6, (v4 + 12) & 0xFFFFFFFC, &v6) < 0 ) return 0;
            v3 += v4 + 5;
            if ( v3 >= v2 ) return v6;
        }
    }
    *v1 = (_WORD)(v3 - v1);
    return v6; }
```

But *v1 is `SMB_FEA_LIST->SizeOfListInBytes`

So if *v1 contains a large value `0x10000` and then we increment
the result

```
*v1 += v3;
```

puts a WORD (16 bits) into what is at address v1

When `SMB_FEA_LIST->SizeOfListInBytes` with incorrect value is used in later code, it can be used to create a buffer overflow, and allows arbitrary code execution...
Spread all over the world in a day
Who is to blame?

- Simple arithmetic mistake
- In a function that is never used (legacy code)
- Who will test this thoroughly?
- But, from wikipedia:

"EternalBlue, sometimes stylized as ETERNALBLUE,\textsuperscript{[1]} is an exploit generally believed to have been developed by the U.S. National Security Agency (NSA). It was leaked by the Shadow Brokers hacker group on April 14, 2017, and was used as part of the worldwide WannaCry ransomware attack on May 12, 2017."

\[\]
Security Testing
Security/penetration testing

- Normal testing investigates correct behavior for sensible inputs, and inputs on borderline conditions.

- Security testing involves looking for the incorrect behavior for really silly inputs.

- Try to crash the system!
  - and discover why it crashed!

- In general, this is very hard.
Why is it hard?

- Systems are (typically) not designed to crash, they work fine on most inputs
- Like finding a needle in a haystack:

![Diagram showing all possible inputs, normal inputs, and an input that triggers a security bug.](image)
Basic technique: random fuzzing

- Test different inputs at random, until the system crashes
- What is the probability of reaching line 11 with random input?

```c
1: int parse(FILE *fp) {
2:     char cmd[256], *url, buf[5];
3:     fread(cmd, 1, 256, fp);
4:     int i, header_ok = 0;
5:     if (cmd[0] == 'G')
6:         if (cmd[1] == 'E')
7:             if (cmd[2] == 'T')
8:                 if (cmd[3] == 't')
9:                     header_ok = 1;
10:    if (!header_ok) return -1;
11:    url = cmd + 4;
12:    i=0;
13:    while ((i<5 && url[i]!='$' && url[i]!='$n') {  
14:        buf[i] = tolower(url[i]);
15:        i++;
16:    }
17:    buf[i] = '\0';
18:    printf("Location is %s\n", buf);
19:    return 0;
```
Structured input

- When input has to start with eg. ‘http’, testing all possible strings that start differently is a waste of time.

- Fortunately, we often know:
  - Example input files or strings
  - Protocol specifications, or test implementations

- Solutions:
  - Generate random permutations from example files
    - Mutation-based fuzzing
  - Fuzz only values but keep in line with the specification
    - Protocol (generative) fuzzing
Mutation-based fuzzing example

1. Google for .pdf
2. Crawl pages to build a test set
3. Use mutation-based fuzzing tool (e.g., ZZuf) or script:
   a) Load pdf file
   b) Mutate file (e.g., randomly flipping bits, adding random chars)
   c) Feed to program
   d) Record if it crashed and what crashed it

A piece of cake, and it can find many real-world bugs!
Mutation-based fuzzing example 2

- Fuzzing with 5 lines of Python code:
  numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) + 1
  for j in range(numwrites):
    rbyte = random.randrange(256)
    rn = random.randrange(len(buf))
    buf[rn] = "%c"%(rbyte)

- Given sufficient time/power this will crash your system!

Code by Charlie Miller
Example: GSM protocol fuzzing

- We can use an universal software radio peripheral (USRP) with open source cell tower software (OpenBTS) to fuzz phones

[Mulliner et al, SMS of Death: from analyzing to attacking mobile phones on a large scale]
[Brinio Hond, Fuzzing the GSM protocol, MSc thesis, Radboud University]
Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and on phones
Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and on phones

  eg possibility to send faxes (!?)

  Only way to get rid if this icon: reboot the phone

you have a fax!
Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and on phones

Fuzzing is a lot of fun!  

eg possibility to send faxes (!?)
Only way to get rid if this icon: reboot the phone
Example: GSM protocol fuzzing

- More serious: malformed SMS text messages display raw memory content, rather than a text message.
AFL and ImageMagick

- AFL is a fast mutation-based fuzzer
  - [http://lcamtuf.coredump.cx/afl/](http://lcamtuf.coredump.cx/afl/)

- Azqa’s fuzzing video:
  - [https://www.youtube.com/watch?v=ibjKz7GTT3I](https://www.youtube.com/watch?v=ibjKz7GTT3I)

- More on:
  - [https://imagetragick.com/](https://imagetragick.com/)
What other information is there?

- We have access the actual system code when testing!

```c
1: int parse(FILE *fp) {
2:     char cmd[256], *url, buf[5];
3:     fread(cmd, 1, 256, fp);
4:     int i, header_ok = 0;
5:     if (cmd[0] == 'G')
6:         if (cmd[1] == 'E')
7:             if (cmd[2] == 'T')
8:                 if (cmd[3] == 'I')
9:                     header_ok = 1;
10:            if (!header_ok) return -1;
11:            url = cmd + 4;
12:            i=0;
13:            while (i<5 && url[i]!='$' && url[i]!='\n') {
14:                buf[i] = tolower(url[i]);
15:                i++;
16:            }
17:            buf[i] = '$';
18:            printf("Location is %s\n", buf);
19:            return 0; }
```

- Can we automatically generate interesting input values?
Code coverage

• Many fuzzing tests will result in the same behavior, to save time, use heuristics!
  • line coverage, statement coverage, branch coverage

• Statement coverage does not imply branch coverage:
  ```c
  void f(int x, y) { if (x>0) {y++;} y--; }
  ```

  statement coverage needs 1 test case
  branch coverage needs 2
Fuzzing heuristics

• To fuzz, you need to select an example input, and apply mutations

• Use code coverage to:
  • Not select an example with coverage identical to selected examples
  • Select examples that add new coverage
  • Apply mutations that led to more coverage
  • ...

• Many fuzzing tools aim to generate new inputs that cover more code, but use different heuristics
  • It pays off to try multiple tools!
Path exploration

- Try to assignments to all values in cmd that make the program reach line 11:
  - Represent all values as symbolic variables
  - Write down a formula describing all paths through the program that reach line 11

**SPECIFY INPUT as symbolic variable:**

<table>
<thead>
<tr>
<th>cmd: cmd0</th>
<th>cmd1</th>
<th>cmd2</th>
<th>cmd3</th>
<th>cmd4</th>
<th>cmd5</th>
<th>cmd6</th>
<th>cmd7</th>
<th>cmd8</th>
<th>cmd9</th>
</tr>
</thead>
<tbody>
<tr>
<td>example:</td>
<td>'G'</td>
<td>'E'</td>
<td>'T'</td>
<td>'h'</td>
<td>'t'</td>
<td>'p'</td>
<td>':'</td>
<td>'/'</td>
<td></td>
</tr>
</tbody>
</table>

(we’re considering input of length 10 just for this example)
Path exploration

SPECIFY INPUT:

```
cmd: cmd0 | cmd1 | cmd2 | cmd3 | cmd4 | cmd5 | cmd6 | cmd7 | cmd8 | cmd9
```

(we're considering input of length 10 just for this example)

SPECIFY PATH CONSTRAINTS:

```
(cmd0 == 'G') & (cmd1 == 'E') & (cmd2 == 'T') & (cmd3 == ' ')
```

FINAL FORMULA:

```
(cmd0 == 'G') & (cmd1 == 'E') & (cmd2 == 'T') & (cmd3 == ' ')
```

```
header_ok = 0;
if (cmd[0] == 'G')
  if (cmd[1] == 'E')
    if (cmd[2] == 'T')
      if (cmd[3] == ' ')
        header_ok = 1;
if (!header_ok)
```
Symbolic execution

- Represent all inputs as **symbolic values** and perform operations symbolically
  - cmd0, cmd1, ...

- Path predicate: is there a value for command such that
  \[(\text{cmd}_0 = \text{'G'}) \& (\text{cmd}_1 = \text{'E'}) \& (\text{cmd}_2 = \text{'T'}) \& (\text{cmd}_3 = \text{' '} \}} \right]?

- Provide all constraints to a combinatorial solver, eg. Z3
  - Answer: YES, with cmd0 = \text{'G'}, cmd1 = \text{'E'}, ..., cmd9 = x

- Only fuzz inputs that satisfy the provided answer!
Symbolic execution, example

```c
m(int x,y){
    x = x + y;
    y = y - x;
    if (2*y > 8) { ....
    }
    else if (3*x < 10){ ... 
    }
}
```

Write down the path predicate needed to reach this line
Symbolic execution, example

```c
m(int x,y){
    x = x + y;  // let x == N and y == M
    y = y - x;  // x becomes N+M
    if (2*y > 8) {... // taken if 2*-N > 8, ie N < -4
        ...
    }
    else if (3*x < 10){... // taken if N>=-4 and 3(M+N)<10
        ...
    }
}

So, (N>=-4) & 3(M+N)<10
```
Not always possible

```javascript
m(function arg) {
  a = 0
  call(arg)
  a = 1
}
```

To determine whether a will ever be 1, one needs to solve the Halting problem...
Not always possible

```
m(function arg)
    a = 0
    call(arg)
    a = 1
```

To determine whether a will ever be 1, one needs to solve the Halting problem...

But used by Microsoft to find and prevent thousands of bugs in Windows!

check: http://www.pexforfun.com
Would security testing have found Heartbleed?

- The root cause is memory management, but it is not a standard buffer overflow since it reads memory instead of writes.

- Why was it not discovered immediately?
  - Only manifests itself on malicious input, works fine normally
  - Does not cause a crash, reads memory from the same process
  - (strange) heartbeat requests are not logged

- Fuzzing will definitely trigger the bug, but since it does not crash, or leave a trace, *it is necessary to also test assertions/logic*
Would security testing have found Stagefright?

• **It did!**

• Using American Fuzzy Lop:
  • By Michal Zalewski "lcamtuf" (Google)
    • [http://lcamtuf.coredump.cx/afl/](http://lcamtuf.coredump.cx/afl/)
  • Mutation based with genetic algorithm
    • Aims to maximize branch-coverage

• run for about 3 weeks, \(\approx\)3200 tests per second
• *Total CPU hours was over 6 months!*
Would security testing have found WannaCry?

• **Probably not**...

• Requires the SMB server to be in a very specific state before the mistake occurs, and then it only leads to an error after additional steps...

• Fuzzers are not (yet) capable of testing this

• **But the tools you learn in this course might be used for this purpose!**
Learning/Reversing
My research

- Traditional
  - code analysis and finding malware fingerprints

- Code/binary analysis is mostly manual and increasingly harder
  - Code obfuscation
  - Encryption
  - Self-modifying

- Behavior-based analysis is much harder to thwart
  - Bots need to communicate!
Learning (reverse-engineering)

- One last piece of information are all the examples that are tested while fuzzing, or collected from logs.
- This form a big data set from which can be used to gain information about a system or protocol.

This can help to:
- analyze your own code and hunt for bugs, or
- reverse-engineer someone else’s unknown protocol, eg. a botnet, to fingerprint or to analyze (and attack) it.
A simple state machine

```c
int current_state = 0;
char step(char input) {
    switch (current_state) {
        case 0:
            switch (input) {
                case 'A':
                    current_state = 1;
                    return 'X';
                case 'B':
                    current_state = 2;
                    return 'Y';
                case 'C':
                    return 'Z';
                default:
                    invalid_input();
            }
        case 1:
            switch (input) {
                case 'A':
                    current_state = 3;
                    return 'Z';
                case 'B':
                    return 0;
            }
        case 2:
            switch (input) {
                case 'A':
                    return 'Z';
                case 'B':
                    return 'Y';
                case 'C':
                    current_state = 0;
                    return 'Z';
                default:
                    invalid_input();
            }
        case 3:
            current_state = 0;
        default:
            return 'Y';
    }
}
```
The same code – obfuscated

```
1__2314 = o__11 != o__20 ? 7 : 10;
while (1) {
    switch (1__2314) {
        case 12:
            o__28(2, o__16);
            1__2314 = 11 - ((o__11 != o__20) + (o__11 != o__28));
            break;
        case 15:
            1__2305 = scanf((char const */* restrict *)
                           (o__19), &1__2303);
            1__2314 = 14 + !(o__11 == o__20);
            break;
        case 2:;
        1__2314 = (unsigned long) (o__20 != (struct t__8 *) 0UL)
            - (unsigned long) (o__11 == (struct t__8 *) 0UL);
            break;
        case 13:
            1__2306 = 1__2307;
            1__2314 = 12 - ((o__20 == (struct t__8 *) 0UL)
                             + (o__20 == (struct t__8 *) 0UL));
            break;
        case 1:
            o__13 = ((1__2304 & ~o__13) << 1) - (1__2304 ^
                              o__13);
            1__2314 = o__20 == (struct t__8 *) 0UL ? 8 : 8;
            break;
        case 3:
```

```
1__2307 = o__12(1__2303);
1__2314 = o__11 == (struct t__8 *) 0UL ? 13 &
1__2304 : 13;
break;
```

```
After learning

```c
int current_state = 0;
char step(char input) {
    switch (current_state) {
        case 0:
            switch (input) {
                case 'A':
                    current_state = 1;
                    return 'X';
                case 'B':
                    current_state = 2;
                    return 'Y';
                case 'C':
                    return 'Z';
                default:
                    invalid_input();
            }
            break;
        case 1:
            switch (input) {
                case 'A':
                    current_state = 3;
                    return 'Z';
                case 'B':
                    return 'Y';
                case 'C':
                    current_state = 0;
                    return 'Z';
                default:
                    invalid_input();
            }
            break;
        case 2:
            switch (input) {
                case 'A':
                    return 'Z';
                case 'B':
                    return 'Y';
                case 'C':
                    current_state = 0;
                    return 'Z';
                default:
                    invalid_input();
            }
            break;
        case 3:
            return 0;
    }
    return 'Z';
}
```
MegaD botnet protocol

Cho et al. 2010
MegaD botnet protocol

Cho et al. 2010

Spamming state
TLS RSA BSAFE

ClientHelloRSA
ServerHello / Certificate / ServerHelloDone

Other
Alert Fatal (Unexpected message) / ConnectionClosed

ClientKeyExchange
Empty

Other
Alert Fatal (Unexpected message) / ConnectionClosed

ChangeCipherSpec
Empty

Other
Alert Fatal (Unexpected message) / ConnectionClosed

ChangeCipherSpec / Finished

Finished

ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed

ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

TU Delft

Joeri de Ruiter & Erik Poll 2015
GNU TLS 3.3.8
Printer controller

Smeenk et al. 2013
Main messages

1. Be careful when programming in C(++)!
2. Never make input assumptions!
3. Test your software for unusual input!
4. Use tools to automate testing!
5. Keep your system up-to-date!

and understand WHY...