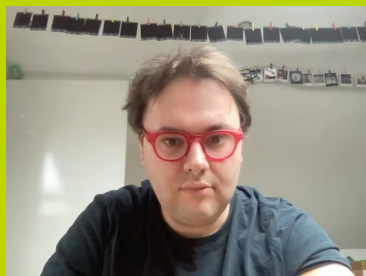
University of BRISTOL

COMS20012: Buffer Overflow

Joseph Hallett



bristol.ac.uk

What's all this about then?

- You get told that functions like `gets` or `strcpy` in C are *dangerous and should not be used*…
- …some OSs will even start outputting warnings to users!?
- …why?

```
$ cat test.c
#include <stdio.h>
int main(void) { char *str; gets(str); return 0; }
$ ./test
warning: this program uses gets(), which is unsafe.
```

What is a buffer overflow?

- What happens when you declare array?
  - You get a region of memory
- Pointers are used to address arrays
  - Very easy to fall off the end of the region!

- Have been known about since the dawn of computers, but earliest tutorial on how to exploit them in *Phrack magazine*

- *Smashing the Stack for Fun and Profit by Aleph1*

`http://phrack.org/issues/49/14.html`

What happens when we call a function?

example1.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

```
bottom of                                                          top of
memory                                                             memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][   ][   ][   ][   ][   ]

top of
stack
```

What about now?

example2.c

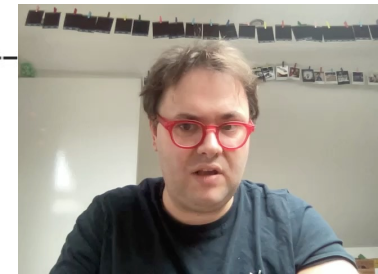
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```



example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
bottom of                                                          top of
memory                                                             memory
                   buffer            sfp    ret    *str
<------           [AAAAAAAAAAAAAAAA][    ][    ][    ]

top of
stack
```

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
bottom of                                                          top of
memory                                                             memory
                   buffer            sfp    ret    *str
<------           [AAAAAAAAAAAAAAAA][AAAA][    ][    ]

top of
stack
```

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
bottom of                                                          top of
memory                                                             memory
                    buffer             sfp    ret    *str
<------           [AAAAAAAAAAAAAAAA][AAAA][AAAA][    ]

top of
stack
```

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
bottom of                                                                top of
memory                                                                   memory
                   buffer            sfp    ret    *str
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][AAAA]AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

top of
stack
```

Why is this bad?

- Lets say this overflow happens…
  - …maybe you corrupt some local stack data
  - …maybe you overflow onto some protected memory region and trigger a segfault?

- Suppose you *don't* trigger a segfault…
  - …what happens when the function returns?

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
bottom of                                                          top of
memory                                                             memory

                  buffer            sfp   ret   *str
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][    ]
                                            |
top of                                      V
stack                                      rip
```

Why is this bad?

- At this point the program *probably* crashes
- Unless 0xAAAA contains valid program code… the CPU can't run from there so you'll *probably* get an illegal instruction exception
- …Probably

Why is this really bad?

- But we kind of know where some stuff is in memory…
  - …the stack (in particular) is fairly predictable
- …and we control what we put into that buffer…
  - …so we *could* put valid instruction sequences into it
- …in which case we could make the program start to run our own code instead of its own…

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

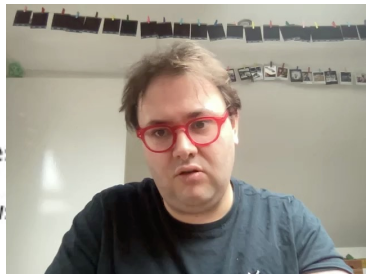
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Introduction

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work.



How do we stop this?

- Modern CPUs don't allow you to write to regions of memory you can execute, or execute from regions of memory you can write to
- But you can get round this…
  - Return to libc or ROP (We'll cover them in Software and Systems Security in year 4)
- Stack canaries help prevent exploitation
  - Stick a random number before the return address… check it hasn't changed before returning
- Shadow stacks also help
  - Keep a second stack with just the return addresses on… check its consistent with the main stack
  - Not implemented everywhere

Maybe just don't overflow buffers?

- If you're using C use the bounded strcpy/gets variants
  - strnpcy is better than strcpy (but integer overflow perils await ;-))
- Use bounded data structures instead of pointers to memory regions
- Stop teaching students about the unsafe stuff and hope they don't look it up?