Low Level Software Security III: Integers, ROP, & CFI
Integer Arithmetic in C

- Quiz time!
- What does this code produce?
  - 100 200 300
  - 100 200 44
  - 100 -56 44
- Depends on how a, b, and c are defined
  - Could be even worse, if a or b is declared as a pointer
    - 100 200 -12
    - 100 200 -124
    - ...

```c
a = 100;
b = 200;
c = a+b;
printf("%d %d %d\n",
       (int)a,
       (int)b,
       (int)c);
```
Integer Overflow/Underflow

- C defines fixed-width integer types (like short, int, long, etc.) that do not always behave like integers from elementary school.

- Because of the fixed width, it is possible for an expression to overflow or underflow.
Integer Overflow/Underflow

- How can this be a problem?
- What if $n$ is too large?
- What if $n$ is negative?

```c
my_type* foo(int n)
{
    my_type *ptr = malloc(n * sizeof(my_type));
    for(int i = 0; i < n; i++)
    {
        memset(&ptr[i], i, sizeof(my_type));
    }
    return ptr;
}
```
Integer Overflow/Underflow

- When a value of an unsigned integer type overflows (or underflows), it simply wraps.
  - As if arithmetic operation was performed modulo $2^{(\text{size of the type})}$.

- Overflow (and underflow) of signed integer types is **undefined** in C.
  - Though most implementations wrap.
Integer Overflow/Underflow

- A common first attempt to check for unsigned overflow looks like this:
  - if (a+b > UINT32_MAX) ...

- What’s wrong with this check?
  - Assume a and b are of type uint32

- Expression a+b cannot hold a value greater than UINT32_MAX
  - a+b == (a+b) mod UINT32_MAX
Checking for Overflow/Underflow

- Unsigned overflow checks should use the complementary operation to the one being checked
  - Subtraction to check for addition overflow
    - if (UINT32_MAX - a < b)
  - Division to check for multiplication overflow
    - if ((0 != a) && (UINT32_MAX / a < b))

- More complex for signed types
  - if(((a>0) && (b>0) && (a > (INT32_MAX-b)))
    || (a<0) && (b<0) && (a < (INT32_MIN-b))))
Integer Overflow/Underflow

- Time for another quiz! How large is:
  - Char
    - At least 8 bits. \texttt{sizeof(char)} == 1
  - Short
    - At least 16 bits
  - Int
    - Natural word size of the architecture, at least 16 bits
  - Long
    - At least 32 bits
  - Long Long
    - At least 64 bits
Integer Type Conversion

- Integer type conversions are yet another common source of security vulnerabilities.
  - Whenever a value is changed from one type to another.

- How are values converted from one type to another? What happens to the bit pattern?
  - Truncation
  - Zero-extension
  - Sign-extension
Integer Type Conversion

- **Truncation** occurs when a value with a wider type is converted to a narrower type.
  
  ```c
  uint32_t j = 0xDEADBEEF;
  uint16_t i = j;
  // i == 0xBeEF
  ```

- When a value is truncated, its high-order bytes are removed so that it is the same width as the narrower type.
Integer Type Conversion

- **Zero-extension** occurs when a value with a narrower, unsigned type is converted to a wider type.

- When a value is zero-extended, it is widened so that it is the same width as the wider type.
  - The new bytes are unset (are zero).

```c
uint16_t i = 0xBEEF;
uint32_t j = i;
// j == 0x0000BEEF
```
Integer Type Conversion

- **Sign-extension** occurs when a value with a narrower, signed type is converted to a wider type.

- When a value is sign-extended, it is widened so that it is the same width as the wider type.
  - If the sign bit of the original value is set, the new bytes are set.
  - If the sign bit of the original value is unset, the new bytes are unset.

- Sign-extension is used for signed types rather than zero-extension because it preserves the value.

```c
int8_t i = -127; // 1000 0001
int8_t j = 127;  // 0111 1111
int16_t ki = i;
int16_t kj = j;
// ki == 1111 1111 1000 0001
// kj == 0000 0000 0111 1111
```
Integer Type Conversion

- The following two slides display reference charts that explain particular integer type conversions.
## Integer Type Conversion

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Method</th>
<th>Lost or Misinterpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>signed char</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
<td>Misinterpreted</td>
</tr>
<tr>
<td>unsigned char</td>
<td>signed short int</td>
<td>Zero-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>unsigned char</td>
<td>signed long int</td>
<td>Zero-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsigned short int</td>
<td>Zero-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>unsigned char</td>
<td>unsigned long int</td>
<td>Zero-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>signed char</td>
<td>Preserve low-order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>signed short int</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
<td>Misinterpreted</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>signed long int</td>
<td>Zero-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>signed char</td>
<td>Preserve low-order byte</td>
<td>Lost</td>
</tr>
</tbody>
</table>
# Integer Type Conversion

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Method</th>
<th>Lost or Misinterpreted</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>short int</td>
<td>Sign-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>signed char</td>
<td>long int</td>
<td>Sign-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>signed char</td>
<td>unsigned char</td>
<td>Preserve bit pattern; high-order bit no longer sign bit</td>
<td>Misinterpreted</td>
</tr>
<tr>
<td>signed char</td>
<td>unsigned short int</td>
<td>Sign-extend to short; convert short to unsigned short</td>
<td>Lost</td>
</tr>
<tr>
<td>signed char</td>
<td>unsigned long int</td>
<td>Sign-extend to long; convert long to unsigned long</td>
<td>Lost</td>
</tr>
<tr>
<td>signed short int</td>
<td>char</td>
<td>Truncate to low-order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>signed short int</td>
<td>long int</td>
<td>Sign-extend</td>
<td>Safe</td>
</tr>
<tr>
<td>signed short int</td>
<td>unsigned char</td>
<td>Truncate to low-order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>signed short int</td>
<td>unsigned short int</td>
<td>Preserve bit pattern; high-order bit no longer sign bit</td>
<td>Misinterpreted</td>
</tr>
<tr>
<td>signed short int</td>
<td>unsigned long int</td>
<td>Sign extend pattern to long; convert long to unsigned</td>
<td>Lost</td>
</tr>
<tr>
<td>signed long int</td>
<td>char</td>
<td>Truncate to low order byte</td>
<td>Lost</td>
</tr>
</tbody>
</table>
Integer Type Conversion

- Type conversion can be explicit or implicit
- Explicit type conversions use the casting syntax:
  - int i = (int) 4.5;
- Implicit type conversions:
  - signed short i = 1;     // assignment conversion
  - unsigned int j = 2;    // assignment conversion
  - if (i < j) { ... }     // comparison conversion
  - void function(int arg);
    function(5.3);         // function argument conversion
Integer Type Conversion

- Rank
  - Rank order (descending):
    - [un]signed long long
    - [un]signed long
    - [un]signed int
    - [un]signed short
    - char, [un]signed char

- Promotion
  - “If a signed int can represent all values of the original type, the value is converted to signed int; otherwise, it is converted to unsigned int”
Integer Type Conversion

- Integer promotions allow operations to be performed in a natural size for an architecture.

- Integer promotions avoid arithmetic errors caused by the overflow of intermediate values.
  - What result do you expect?
  - What result would you get without integer promotion?
  - Thus, integer promotions also help operations produce the expected, intuitive result by avoiding overflow.

```c
signed char a = 100;
signed char b = 3;
signed char c = 4;
signed char result = a*b / c;
```
Integer Type Conversion: Common Real Type

- Are both operands of the same type?
  - No further conversion is needed.

- If not, are both operand signed or both unsigned?
  - The operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

- If not, is rank of unsigned operand >= rank of signed operand?
  - The operand with signed integer type is converted to the type of the operand with unsigned integer type.

- If not, can the type of the signed operand represent all of the values of the type of the unsigned operand?
  - The operand with unsigned integer type is converted to the type of the operand with signed integer type.

- If not, both operands are converted to the unsigned integer type.
Integer Type Conversion Simplified

- Need to determine rank and signedness

- CRT is signed if and only if:
  - Both operands are signed
  - Or one operand is signed and size of signed operand type is larger than size of unsigned operand type

- CRT.rank = max(a.rank, b.rank, int.rank)
### Integer Type Conversion

- What type are integer constants (literals)?

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal Constant</th>
<th>Octal or Hexadecimal Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>
Integer Type Conversion Example 1

- Let’s go through an example:
  - unsigned long a;
  - signed short b;
  - a + b;

- Both operands have rank >= int?
  - No. b has to be promoted to signed int.
  - Because b is signed, sign-extension is used.

- Both operands of the same type?
  - No. Need to convert to common real type.

- Both operands signed or both unsigned?
Rank of unsigned operand $\geq$ rank of signed operand?

- Yes. The operand with signed integer type is converted to the type of the operand with unsigned integer type.
- b is sign-extended to width of long and then the bit pattern is re-interpreted as an unsigned value.
- Note that this could result in a negative number being interpreted as a very large positive number.
Let’s go through another example:

Both operands have rank >= int?
  - Yes.

Both operands of the same type?
  - No. Need to convert to common real type.

Both operands signed or both unsigned?
  - No.
Integer Type Conversion Example 2

- Rank of unsigned operand >= rank of signed operand?
  - No.

- Can the type of signed operand represent all values of the type of unsigned operand?
  - Yes. The operand with unsigned integer type is converted to the type of the operand with signed integer type.
  - b is zero-extended to width of long long and then the bit pattern is re-interpreted as a signed value.

```c
signed long long a;
unsigned int b;

a + b;
```
What does this code print?

Let’s walk through conversion to common real type...

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```
Integer Type Conversion

- Both operands have rank $\geq \text{int}$?
  - Yes.

- Both operands of the same type?
  - No.

- Both operands signed or both unsigned?
  - No.

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```
Integer Type Conversion

- Rank of unsigned operand >= rank of signed operand?
  - Yes.
  - Then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
  - `(unsigned int)-1 == UINT_MAX`

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```
Integer Type Conversion

- Take a moment to review this code.

- Note the check to verify that port number is >= 1024 unless the user is root.
  - The first 1024 ports are assigned to critical network services, such as HTTPS, POP, and SMTP and are considered privileged.

```c
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
    //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
   // handle error
else
    sockaddr.sin_port = port;
```
Integer Type Conversion

- The field `sin_port` is declared as a 16-bit unsigned integer.
  - Range \([0, 2^{16} - 1]\).

- The variable `port` is declared as a 32-bit signed integer.
  - Range \([-2^{31}, 2^{31} - 1]\).

- When `port` is assigned to `sin_port`, the two high-order bytes of the value are truncated and the port number is changed.

```c
struct sockaddr_in
{
    short    sin_family;
    u_short  sin_port;
    //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
    // handle error
else
    sockaddr.sin_port = port;
```
Integer Type Conversion

▪ What happens if an attacker sets the variable `port` to 65979?

▪ The comparison within the if statement is between two signed 32-bit integers and 65979 is greater than 1024.

▪ But, note the hexadecimal representation of 65979 – 0x000101BB

▪ When `port` is assigned to `sin_port`, its value is truncated, and `sin_port` is set to 0x01BB, or 443.

```c
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root) // handle error
else
    sockaddr.sin_port = port;
```
Integer Type Conversion

- Take a moment to review this code.
- Prototype for `read()`:
  ```c
  size_t read(
    int fildes,
    void* buf,
    size_t nbyte);
  ```

```c
int sockfd;

int length;
char buffer[1204];

length = get_user_length(sockfd);

if(length > 1024)
{
    error("illegal input, not enough room in buffer\n");
    return -1;
}

read(sockfd, buffer, length);
```
Integer Type Conversion

- What is the type of `length`?
- What is the type of the third formal argument to `read()`?
  - `size_t` is a 32-bit unsigned integer.
- A negative value for `length`, such as `-1 (0xFFFFFFFF)` will be interpreted as a very large positive integer.

```c
int sockfd;
//...
int length;
char buffer[1204];

length = get_user_length(sockfd);
// reads a 32-bit int from the network
if(length > 1024)
{
    error("illegal input, not "
          "enough room in buffer\n");
    return -1;
}
read(sockfd, buffer, length);
```
Integer Type Conversion

- To address this vulnerability, `length` should be declared as `size_t`.

```c
int sockfd;
//...
size_t length;
char buffer[1204];

length = get_user_length(sockfd);
// reads a 32-bit int from the network
if (length > 1024)
{
    error("illegal input, not "
          "enough room in buffer\n");
    return -1;
}
read(sockfd, buffer, length);
```
Integer Type Conversion

- So, how hard is it to correctly add two numbers in C?
Integers

- Before you decide that C has a monopoly on arithmetic madness...

- Some JavaScript fun:

  ```javascript
  3   - 1  // -> 2
  3   + 1  // -> 4
  '3' - 1  // -> 2
  '3' + 1  // -> '31'
  [1, 2, 3] + [4, 5, 6]  // -> '1,2,34,5,6'
  1 < 2 < 3  // -> true
  3 > 2 > 1  // -> false
  ```
All 0s Are Equal

- So, PHP...

- Remember scientific notation?
  - Example: \(1.2e3 == 1.2 \times 10^3\)

- And if we have \(0e[\text{anything at all}]\)?
  - \(0 \times 10^[\text{anything at all}] == 0\)

- And if we have a long hexadecimal string (like a hash) that happens to start with \(0e\), and we want to compare it to another long hexadecimal string to make sure they are equal?
  - If they both start with “0e”, then they are equal!
  - \(0 == 0\)
Return Oriented Programming
Bypassing Code Injection Mitigations

- From last time...

- Data Execution Prevention
  - If we prevent attackers from injecting code, we deny them ability to execute arbitrary code.
  - All pages are either writeable or executable, but not both.
  - We won! ... right?

- Is there another way for an attacker to execute arbitrary code even without the ability to inject it into the victim process?
Code Reuse Attacks

- What do you do if you can’t inject new code?
- Use the code that’s already there.
- What code is already there?
  - Program + shared libraries (including libc)
Return To Libc

- What can we find in libc?
  - “The `system()` library function uses `fork(2)` to create a child process that executes the shell command specified in `command` using `execl(3)` as follows:
    
    ```c
    execl("/bin/sh", "sh", ",c", command, (char *) 0);
    ```
  
  - Need to find the address of:
    - `system()`
    - String “/bin/sh”
  
  - Overwrite the return address to point to start of `system()`
  
  - Place address of “/bin/sh” on the stack so that `system()` uses it as the argument
Return To Libc

- Many different variants

- What else can attackers do by calling available functions with parameters of their choosing?
  - Move shellcode to unprotected memory.
  - Change permissions on stack pages.
  - Etc.

Another CTF trick: if you need a string for system() that will get you a shell, consider the humble "ed". It supports running shell commands (!), and b/c of English past tense is often available as a suffix of some existing string in the binary, e.g.: "File transfer completed"
Employees must wash hands before returning to libc

@duosec
Return Oriented Programming (ROP)

- What if we cannot find just the right function?
- What happens if we jump almost to the end of some function?
  - We execute the last few instructions, and then?
  - Then we return. But where?
  - To the return address on the stack. But we overwrote the stack with our own data.
  - Let’s return to another function tail.
  - Rinse and repeat.
Return Oriented Programming (ROP)

- ROP idea: make shellcode out of existing application code.

- Stitching together arbitrary programs out of code gadgets already present in the target binary
  - **Gadgets**: code sequences ending in ret instruction.
  - May be deliberately added by compiler (at end of function)
  - Or any sequence in executable memory ending in \(0xC3\).
    - x86 has variable-length instructions.
    - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences.

- Overwrite saved return address on stack to point to first gadget, the following word to point to second gadget, etc.

- Stack pointer is the new instruction pointer.
“Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.”

- The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) by Hovav Shacham
  - https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

- Turing-complete computation.
  - Load and Store gadgets
  - Arithmetic and Logic gadgets
  - Control Flow gadgets
Return Oriented Programming

- Gadget for loading a constant
  - Arrange the constant to load to be just past the return address
  - Return to gadget that pops a value and returns.
Return Oriented Programming

- Gadget for loading a value from memory
  - A bit more complex...
  - Arrange the address* of value to be loaded past the return address
    - *or address-immediate
  - Return to gadget that pops that address into %eax
  - Return to gadget that loads the value based on address in %eax
Return Oriented Programming

- Gadgets for arithmetic operations get more complex
  - Addition is one of the simpler arithmetic gadgets

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming
Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>16</td>
</tr>
</tbody>
</table>

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

<table>
<thead>
<tr>
<th>step</th>
<th>ecx</th>
<th>edx</th>
<th>edi</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>esp[4]</td>
<td>esp+32</td>
<td>~</td>
<td>16</td>
</tr>
</tbody>
</table>
Return Oriented Programming

- **Control Flow Gadgets**
  - Stack pointer is effectively the new instruction pointer
  - To “jump” just pop a new value into `%esp`.
  - Conditional jumps are more involved but still possible

![Diagram of control flow gadgets](https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf)
Return Oriented Programming

- Stack pointer acts as instruction pointer
- Manually stitching gadgets together gets tricky
  - Automation to the rescue!
  - Gadget finder tools, ROP chain compilers, etc.
Return Oriented Programming

- A related concept is Jump Oriented Programming
  - Identify gadgets ending in indirect jumps.
  - Use a “dispatcher gadget” to combine them.
  - Dispatch table used in place of stack
Resources

Smashing The Stack For Fun And Profit
Getting Around Non-Executable Stack (and Fix)
The Advanced Return-into-lib(c) Exploits
msfrop / ROPgadget / ropper
Q: Exploit Hardening Made Easy
Resources

Return-Oriented Programming
Jump-Oriented Programming
Call-Oriented Programming
Blind Return-Oriented Programming
Sigreturn-Oriented Programming
Counterfeit-Object-Oriented Programming