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

```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 (*short*, *int*, *long*, etc.) that do not always behave like Platonic integers from elementary school.

- Because of the fixed width, it is possible to overflow or wrap maximum expressible number for the type used
  - Or underflow in case of negative numbers
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;
}
```
Classic overflow example

```c
void *ConcatBytes(void *buf1, unsigned int len1,
                 char *buf2, unsigned int len2)
{
    void *buf = malloc(len1 + len2);
    if (buf == NULL) return;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
}
```

What if:
len1 == 0xFFFFFFFFE
len2 == 0x000000102

ox100 bytes allocated... not enough. Ooops.
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 need to 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. `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 32 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.

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

```c
uint32_t j = 0xDEADBEEF;
uint16_t i = j;
// i == 0x00BEEF
```
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>
<tr>
<td>unsigned long int</td>
<td>signed short int</td>
<td>Preserve low-order word</td>
<td>Lost</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>signed long int</td>
<td>Preserve bit pattern; high-order bit becomes sign bit</td>
<td>Misinterpreted</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>unsigned char</td>
<td>Preserve low-order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>unsigned short int</td>
<td>Preserve low-order word</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 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>
<tr>
<td>signed long int</td>
<td>short int</td>
<td>Truncate to low order bytes</td>
<td>Lost</td>
</tr>
<tr>
<td>signed long int</td>
<td>unsigned char</td>
<td>Truncate to low order byte</td>
<td>Lost</td>
</tr>
<tr>
<td>signed long int</td>
<td>unsigned short int</td>
<td>Truncate to low order bytes</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

- Implicit conversions are common in arithmetic expressions.
- Prior to evaluation within an expression, numeric operands are converted:
  - First, they are “promoted” to be of at least int rank
    - 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
  - Rank order (descending):
    - [un]signed long long >
    - [un]signed long >
    - [un]signed int >
    - [un]signed short >
    - char, [un]signed char
  - Then they are converted to Common Real Type
- Conversion rules are complex
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 corresponding to the type of the operand with signed integer type.
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></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long 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></td>
<td>unsigned long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned 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>
Why?!?!?

- Integer promotions allow operations to be performed in a natural size for an architecture.
  - The `signed int` and `unsigned int` types correspond to the natural word size of an architecture.
  - Most efficient to perform operations in the natural size of an architecture.

- Integer promotions avoid arithmetic errors caused by the overflow of intermediate values.
  - For example, consider this code.
  - 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.

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

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

- Both operands have rank $\geq$ `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?
  - No.
Integer Type Conversion

- 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 $\text{long}$ 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.

```c
unsigned long a;
signed short b;
a + b;
```
Integer Type Conversion

- The rules for type conversions are complex and not always intuitive.
- It is not always obvious where type conversions occur.
- Another quiz! 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 \(\geq\) 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 $\geq 1024$ 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 < 1204 && !is_root)
    // handle error
else
    sockaddr.sin_port = port;
```
Integer Type Conversion

- Where is the vulnerability?
- Is there an input that will allow a normal user to open a connection on a privileged port?

```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.

- Port 443 is used for HTTPS traffic.

```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

- Conversions between signed and unsigned types can lead to exploitable vulnerabilities.

- For example, using signed integers for length values and not considering the potential for negative values.
  - When a signed negative value is converted to an unsigned variable of the same size, the bit pattern is left alone, but the value changes.
  - This type of bug happens commonly when a maximum length check is performed on a user-supplied integer, but no check is made to see whether the integer is negative.
What to do?

- **Strongly typed language**
  - Most such problems go away in Java for example

- **Runtime checking**
  - `gcc -ftrapv` (trap on signed overflow on add, sub, mult)
  - Safe libraries (David LeBlanc’s SafeInt class)
    - Template overrides all operators (not the fastest)
    - Use when variable can be influenced by untrusted input

- **Static checking (range analysis)**
  - Sarkar et al, “Flow-insensitive Static Analysis for Detecting Integer Anomalies in Programs”
Using SafeInt

```c
void *ConcatBytes(void *buf1,
                  SafeInt<int> len1,
                  char *buf2,
                  SafeInt<int> len2)
{
    void *buf = malloc(len1 + len2);
    if (buf == NULL) return;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
}
```

Overload “+”
Will throw exception on overflow
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?
  – Assumption is that you need malicious code for malicious computation. **Not true.**

▪ 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.

Brendan Dolan-Gavitt
@monkeyx

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"
Return Oriented Programming (ROP)

- What if we cannot find just the right function? Or need more complex computation?

- 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.
Aside: how Intel variable length instructions help ROP

- X86 instructions are variable length, yet can begin on any byte sequence

- Example:

  81 c4 88 00 00 00 add $0x00000088, %esp
  5f                  pop %edi
  5d                  pop %ebp
  c3                  ret

  00 5f 5d ad db      addb %bl, 93 (%edi)
  c3                  ret

- Result: more “function tails” to choose from
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.
Return Oriented Programming

▪ “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

- 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
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.
- Also: not even really about “returns”... other variants target other kinds of deterministic control flow
- Well, heck.... what to do?
Control Flow Integrity
Control Flow Integrity

- Given a new attack technique, we must present a new countermeasure.
  - Such is the cycle of life.

- **Control Flow Integrity (CFI)**
  - Constraining the control-flow to only legitimate paths determined in advance.
  - Basic idea: match jump, call, and return sites to their target destinations.
  - Many different implementations with varying degrees of applicability, protection strength, and performance overhead.
Control Flow Integrity

- Focus is on protecting indirect transfer of control flow instructions.

  - **Direct** control flow transfer:
    - Advancing to next sequential instruction
    - Jumping to (or calling a function at) an address hard-coded in the instruction

  - **Indirect** control flow transfer
    - Jumping to (or calling a function at) an address in register or memory
    - **Forward path**: indirect calls and branches.
    - **Reverse path**: return addresses on the stack.
void sort2( int a[],
          int b[],
          int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
Control Flow Integrity

- Basic Design:
  - Assign **labels** to all indirect jumps and their targets
  - After taking an indirect jump, validate that target label matches jump site
    - Like stack canaries, but for control flow targets
  - Hardware support is essential to make enforcement efficient
  - Absent that, make performance/precision tradeoffs
Abadi CFI

- Fine Grained CFI
- Control-Flow Integrity: Principles, Implementations, and Applications
- Statically compute a control flow graph.
- Dynamically ensure that the program never deviates from this graph.
  - Assign a label to each destination of indirect control-flow.
  - Instrument indirect control-flow transfers with checks that compare the label at the destination with a constant value to ensure that the branch target is valid.
- Policy is enforced at every indirect control-flow transfer.
- Ideal (fine-grained) CFI versus practical (coarse-grained) CFI
- Suggested that two labels may be sufficient
- Suggested use of a shadow stack
Control Flow Integrity

```c
void sort2( int a[],
            int b[],
            int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y)
{ return x < y; }

bool gt(int x, int y)
{ return x > y; }
```
CFI: Example of Labels

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx ; computed jump</td>
<td>SB 44 24 04</td>
<td>mov eax, [esp+4] ; dst</td>
</tr>
</tbody>
</table>

**Instrumented code**

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary
Control Flow Integrity

- Coarse-grained CFI
  - Example: *bin-CFI: Control Flow Integrity for COTS Binaries*
    - [https://www.usenix.org/node/174767](https://www.usenix.org/node/174767)

- Two labels
  - One label for destinations of indirect calls
    - Make sure that every call actually lands at function start
  - One label for destinations of return and indirect jump instructions
    - Make sure every indirect jump lands at the start of a basic block

- Policy is enforced at every indirect control-flow transfer.
Control Flow Integrity

```c
void sort2( int a[],
            int b[],
            int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y)
{ return x < y; }

bool gt(int x, int y)
{ return x > y; }
```
CFI Limitations

- **Overhead**
  - Additional computation is needed before every free branch instruction.
  - Additional code size is needed before every free branch instruction and at each location (the label).

- **Scope**
  - Data is not protected.
  - CFI does not protect against interpreters.
  - Needs reliable DEP (if you can change code all bets are off)

- **Precision**
  - If you don’t bind flow for all control transfers, can still create gadgets
  - *Out of Control: Overcoming Control-Flow Integrity*
  - *Stitching the Gadgets: On the Ineffectiveness of Coarse Grained CFI*
  - *Aside: C++ is a big problem due to virtual functions*
CFI Limitations

- *Out of Control: Overcoming Control-Flow Integrity*
- *Stitching the Gadgets: On the Ineffectiveness of Coarse Grained CFI*

**Summary**
- Precision is important.
- Most implementations of CFI trade precision for performance. They restrict an attacker, but the restrictions are not sufficient to stop motivated attackers.
- Return path has a lot more gadgets.
Review

- Integers on computers are not integers
  - Reasoning about computer arithmetic is hard
  - If you use those integers for any sensitive question you can create vulnerabilities

- Return-oriented Programming
  - You can synthesize malicious computation out benign code, but stitching it together with control flow
  - The stack is a convenient place to do this stitching

- Control-flow integrity
  - Promise to provide general-purpose protection against control flow vulnerabilities
  - Not there year: overhead and the precision issues
For next time

- Read Understanding glibc malloc" by sploitfun
Next Lecture...

Low Level Software Security IV: Heap Corruption
(the last low level software security lecture!)