CMPSC 497
Execution Integrity

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Exploit Vulnerabilities

- How do you exploit a memory error vulnerability?
Memory Error Exploits

- First and most common way to take control of a process
- Write to control memory
  - Call the victim with inputs necessary to overflow buffer or exploit data pointer
  - To overwrite the value of a pointer to code (e.g., return address) or data that impacts control (e.g., conditional)
- Direct the process execution to exploit code
  - Inject code (if possible) or reuse existing code
  - Use compromised pointer to jump to the chosen code
Determine what to attack

- Local variable that is a char buffer
  - Called buf

...  
printf("BEFORE picture of stack\n");
for ( i=((unsigned) buf-8); i<((unsigned) ((char *)&ct)+8); i++ )
  printf("%p: 0x%x\n", (void *)i, *(unsigned char *) i);

/* run overflow */
for ( i=1; i<tmp; i++){
  printf("i = %d; tmp= %d; ct = %d; &tmp = %p\n", i, tmp, ct, (void *)&tmp);
  strcpy(p, inputs[i]);

  /* print stack after the fact */
  printf("AFTER iteration %d\n", i);
  for ( j=((unsigned) buf-8); j<((unsigned) ((char *)&ct)+8); j++ )
    printf("%p: 0x%x\n", (void *)&j, *(unsigned char *) &j);

  p += strlen(inputs[i]);
  if ( i+1 != tmp )
    "p++ = '\';
}
printf("buf = %s\n", buf);
printf("victim: %p\n", (void *)&victim);
return 0;
Configure Attack

- Configure following
  - Distance to from buffer to target (return address)
    - Where to write?
  - Location of start of attacker-chosen code
    - Where to direct?
  - What to write on stack
    - How to invoke – if need arguments and/or multiple steps?
  - How to craft the attack
    - How to produce and send the malicious buffer to the victim?
Find Return Address

- **x86 Architecture**
  - Build 32-bit code for Linux environment
- **Remember integers are represented in “little endian” format**
- **Take address 0x8048471**
  - See trace at right
- **How do you know it’s a return address?**
  - Run objdump –dl on the victim, and you will see this is the address after a call instr...
Overwrite Return Address

- Build victim for debugging
  - You may have source code
  - But can debug binary with more powerful tools
- Run the victim under *gdb* to determine addresses and relative offsets
  - Even with ASLR on, the offsets will be the same
  - Craft buffer to write up to the target

---

BEFORE picture of stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0fa3b854</td>
<td>0x3</td>
</tr>
<tr>
<td>0x0fa3b855</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b856</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b857</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b858</td>
<td>0x3</td>
</tr>
<tr>
<td>0x0fa3b859</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85a</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85b</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85c</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85d</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85e</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b85f</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b860</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b861</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b862</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b863</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b864</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b865</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b866</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b867</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b868</td>
<td>0xa8</td>
</tr>
<tr>
<td>0x0fa3b869</td>
<td>0xb8</td>
</tr>
<tr>
<td>0x0fa3b86a</td>
<td>0xa3</td>
</tr>
<tr>
<td>0x0fa3b86b</td>
<td>0xbf</td>
</tr>
<tr>
<td>0x0fa3b86c</td>
<td>0x71</td>
</tr>
<tr>
<td>0x0fa3b86d</td>
<td>0x84</td>
</tr>
<tr>
<td>0x0fa3b86e</td>
<td>0x4</td>
</tr>
<tr>
<td>0x0fa3b86f</td>
<td>0x8</td>
</tr>
<tr>
<td>0x0fa3b870</td>
<td>0x3</td>
</tr>
<tr>
<td>0x0fa3b871</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b872</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0fa3b873</td>
<td>0x0</td>
</tr>
</tbody>
</table>

buf

ebp

rtn addr

ct
Objdumap to Find Code Ptrs

- A return address should point to the instruction after a ‘call’
- May also want to compromise a function pointer - an address in code

```
mov 8 8b45 c0 00, mov -0x20(%rbp),%rax
401a3a:
401a3e:
401a41:
401a46:
401a4b:
401a4e:
401a53:
401a55:
401a5f:
401a63:
401a66:
401a69:
401a6e:
401a70:
401a72:
401a76:
401a7a:
401a7d:
401a81:
401a85:
401a88:
401a8b:
401a8e:
401a93:
401a97:
401a99: 89 45 f4, mov %eax,0xc(%rbp)
401a9d: 83 7d f4 00, cmpl 88x0,%ecx(%rbp)
```

- BEFORE picture of stack
  - 0xbfa3b854: 0x3
  - 0xbfa3b855: 0x0
  - 0xbfa3b856: 0x0
  - 0xbfa3b857: 0x0
  - 0xbfa3b858: 0x3
  - 0xbfa3b859: 0x0
  - 0xbfa3b85a: 0x0
  - 0xbfa3b85b: 0x0
  - 0xbfa3b85c: 0x0
  - 0xbfa3b85d: 0x0
  - 0xbfa3b85e: 0x8
  - 0xbfa3b85f: 0x0
  - 0xbfa3b860: 0x0
  - 0xbfa3b861: 0x0
  - 0xbfa3b862: 0x0
  - 0xbfa3b863: 0x0
  - 0xbfa3b864: 0x0
  - 0xbfa3b865: 0x0
  - 0xbfa3b866: 0x0
  - 0xbfa3b867: 0x0
  - 0xbfa3b868: 0xa0
  - 0xbfa3b869: 0x8b
  - 0xbfa3b86a: 0x0
  - 0xbfa3b86b: 0x0
  - 0xbfa3b86c: 0x71
  - 0xbfa3b86d: 0x84
  - 0xbfa3b86e: 0x4
  - 0xbfa3b86f: 0x8
  - 0xbfa3b870: 0x0
  - 0xbfa3b871: 0x0
  - 0xbfa3b872: 0x0
  - 0xbfa3b873: 0x0

- buf
- ebp
- rtn addr
- ct
Direct to Adversary Code

- How do you know what to change the return address to?
- Can use ‘objdump’ again to find the desired instruction(s)
  - Can call code you want to run that’s already there
  - Can call a library function that’s available
    - Need to fix stack with arguments to function
  - Can create and invoke a ROP gadget chain
    - Need to craft the entire stack you want
Objdump to Find Attack Code

- May want to find particular statements or PLT entries

  - What is “system” good for?

- Or jump directly to executable statement with desired effect

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

int main()
{
    char *buf = NULL;
    unsigned int tmp = 0;
    int i = 1;
    char *p = NULL;
    char *inputs[] = { "hello", "world", "overflow" };
    char *victim = NULL;

    printf("BEFORE picture of stack\n");
    for (i = 0; i < 10; i++)
    {
        printf("%p: 0x%x\n", (void *)i, *((unsigned char *)i));
    }

    strcpy(p, inputs[i]);
    printf("AFTER iteration %d\n", i);
    for (j = 0; j < 10; j++)
    {
        printf("%p: 0x%x\n", (void *)j, *((unsigned char *)j));
    }
    return 0;
}
```

BEFORE picture of stack

```
0xbfa3b854: 0x3
0xbfa3b855: 0x0
0xbfa3b856: 0x0
0xbfa3b857: 0x0
0xbfa3b858: 0x3
0xbfa3b859: 0x0
0xbfa3b85a: 0x0
0xbfa3b85b: 0x0
```

buf

ebp

rtn addr

c

Exploits

- Run code determined by attacker

- Old way
  - Include attack code in malicious buffer value
  - Prevented by modern defenses: NX and randomized stack base

- Modern way
  - Return-to-libc attack
  - Configure the stack to run code in the victim’s code segment

- Return-oriented programming
Prevent Overflows

- Besides using safe string functions (not all buffers are for strings), how would you prevent adversaries from causing overflows?
Prevent Overflows

- Besides using safe string functions (not all buffers are for strings), how would you prevent adversaries from causing overflows?
  - Ensure that all writes using a pointer to buffer memory are within that buffer memory (spatial memory safety)
    - Ensuring all pointers point to allocated memory (or NULL) is called temporal memory safety
  - Two ways to achieve spatial memory safety:
    - Bounds checks
    - Fat pointers with bounds information
Check Bounds

- How would you check bounds naively?
Check Bounds

- How would you check bounds naively?
  - Presumably, you need to know the start and end of a buffer
- Then, you need to check bounds – how and when?
Bounds Checks

- SoftBound
  - Records base and bound information for every pointer as disjoint metadata
  - Check and/or update such metadata whenever one dereferences (uses) a pointer
  - Supported by formal proofs of spatial memory safety

- Separating metadata from pointers maintains compatibility with C runtime
SoftBound

- Checking Bounds
  - Whenever a pointer is used to access memory (i.e., dereferenced), SoftBound inserts code (highlighted in grey) for checking the bounds to detect spatial memory violations.

```c
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
value = *ptr;  // original load
```

Where `check()` is defined as:

```c
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```
SoftBound

- Need to initialize and maintain bounds information
  - How to create?
  - How to lookup bounds info?
  - What ops require changes to bounds info?
Creating pointers

New pointers in C are created in two ways:

- (1) explicit memory allocation (i.e. `malloc()`)
- (2) taking the address of a global or stack-allocated variable using the ‘&’ operator.

Initialization for `malloc`

```c
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```
• Pointer metadata retrieval
  ‣ SoftBound uses a table data structure to map an address of a pointer in memory to the metadata for that pointer
  ‣ On load
    
    ```c
    int** ptr;
    int* new_ptr;
    ...
    check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
    newptr = *ptr; // original load
    newptr_base = table_lookup(ptr)->base;
    newptr_bound = table_lookup(ptr)->bound;
    ```
  ‣ On store
    
    ```c
    int** ptr;
    int* new_ptr;
    ...
    check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
    (*ptr) = new_ptr; // original store
    table_lookup(ptr)->base = newptr_base;
    table_lookup(ptr)->bound = newptr_bound;
    ```
• Pointer arithmetic

  ‣ When an expression contains pointer arithmetic (e.g., `ptr+index`), array indexing (e.g., `&(ptr[index])`), or pointer assignment (e.g., `newptr = ptr;`), the resulting pointer inherits the base and bound of the original pointer

  ```c
  newptr = ptr + index;  // or &ptr[index]
  newptr_base = ptr_base;
  newptr_bound = ptr_bound;
  ```
SoftBound

• Downsides
  ‣ Has a significant overhead – 67% for 23 benchmark programs
  ‣ Uses extra memory – 64% to 87% depending on implementation
  ‣ Does not support multithreaded programs

• But, achieve full spatial memory safety for C programs without modifications for benchmarks
  ‣ We use in “privilege separation” work to be discussed later
Fat Pointers

- **Idea**
  - Associate base and bounds metadata with every pointer

- **Problems**
  - **Forgery** – overwrite base and bounds when overwrite pointer
  - **Limited space** – have at most 64 bits to express
  - **Performance** – SoftBound demonstrated that these operations could be costly

- **Solutions?**
Low-Fat Pointers

• Idea
  ‣ Hardware support for fat pointers

• Solutions
  ‣ Forgery – Hardware tags to prevent software from overwriting without detection
  ‣ Limited space – Do not really need entire 64-bit address space – use 46-bit address space and rest for metadata
  ‣ Performance – Hardware instructions to perform desired operations inline

• Result: Memory error protection for 3% overhead
Low-Fat Pointers

- Checking – similar to SoftBound

```c
if ((ptr.A >= ptr.base) && (ptr.A <= ptr.bound))
    perform load or store
else
    jump to error handler
```

- Tagging – common technique from long ago
  - Hardware differentiates data (and code) from references
  - Utilize 8 bits of 64-bit pointer for “type” of pointer

- Encoding
  - Base and bounds within the rest
Low-Fat Encoding

- Challenge
  - Could require at least two pointers
  - So how to encode in a few bits?
Low-Fat Encoding

• Challenge
  ‣ Could require at least two pointers
  ‣ So how to encode in a few bits?

• Restrictions
  ‣ Size of buffer is a power of 2
  ‣ Pointer is aligned on a power of 2 boundary

• Determine base and bounds
  ‣ **Base:** Set n of the LSB’s to zero and store n in $\log_2 n$ bits
  ‣ **Bounds:** Set m of the LSBs to one to find bound
Low-Fat Encoding

- Determine base and bounds
  - **Base**: Set \( n \) of the LSB’s to zero and store \( n \) in \( \log_2 n \) bits
  - **Bounds**: Set \( m \) of the LSBs to one to find bound

- 0x12345678
  - Assume \( n \) is 8
  - What is base address?
  - Assume \( m \) is 5
  - What is bound length? And length?
Low-Fat Encoding

- Determine base and bounds
  - **Base**: Set \( n \) of the LSB’s to zero and store \( n \) in \( \log_2 n \) bits
  - **Bounds**: Set \( m \) of the LSBs to one to find bound

- 0x12345678
  - Assume \( n \) is 8
  - What is base address? 0x12345600
  - Assume \( m \) is 5
  - What is bound length? And address?
    - \( 2^5 \) and 0x1234561F
Low-Fat Encoding Hardware

• OK, great – we can conceivably have bounds information encoded in pointers
  ‣ But we need specialized hardware

• Two solutions
  ‣ Make tagged references part of hardware you use – investigation of such hardware architectures (CHERI), which we’ll discuss later
  ‣ Encode meta-data checks into code with pointers, kind of like SoftBound
    • More efficient checking methods are under development
Direct Control of Program

- Once an adversary can specify the value of a pointer used in control flow, they can direct the program’s execution
  - Return address (call stack) – choose next code to run on return instruction
  - Function pointer – chooses next code to run when invoked

- How do limit the exploit options available to adversaries?
Prevent ROP Attacks

• Most powerful adversary attack is ROP
• Using a ROP chain can execute any code in any order
  ‣ As long as it terminates in a return instruction
  ‣ Can also chain calls and jumps
• How would you prevent a program from executing the victim’s code in unexpected and arbitrary ways?
Prevent ROP Attacks

- How would you prevent a program from executing gadgets rather than the expected code?
  - Control-flow integrity
    - Force the program to execute according to an expected CFG
Control Flow Graph

- Is a graph $G=(V,E)$
  - Graph vertices: $V$ – set of program instructions
  - Graph edges: $E=(a, b)$ – meaning $b$ can succeed $a$ in some execution

- For a function a CFG relates the instructions and the possible ordering of instruction executions

- Many of these can be predicted from the code
Control Flow Graph

- Each line corresponds to one or more instructions
- Non-trivial edges
  - Line 1 → 11
  - Line 3 → 5
  - Line 7 → 9
- All flow edges known from code

0: /* i, n are ints, and char b[12] */
1: if (i > 0) {
2:     n = i + 2;
3:     if (n == 7)
4:         b[n+i] = 'a';
5:     else {
6:         n = i + 8;
7:         if (n < 12)
8:             b[n] = 'a';
9:         }
10: }

(a) (3pts) Specify the symbolic path constraints in terms of i for lines 3, 5, and 7.
(b) (3pts) How can knowledge of these symbolic path constraints aid in fuzz testing?
(c) (2pts) Is there an error in this program?
CFG Ambiguity

- There is ambiguity about the target of some instructions
  - Called indirect control flows
- Those instructions are
  - Returns
  - Indirect Calls
  - Indirect Jumps
- Their targets are computed at runtime
  - Can you give an example?
Control-Flow Integrity

Our Mechanism

- **FA**: 
  - **nop IMM₂**
  - **call fp**
  - `if(*fp != nop IMM₂) halt`

- **FB**: 
  - **nop IMM₁**
  - `if(**esp != nop IMM₂) halt`
  - **return**

CFG excerpt

- `A_{call}`
- `B₁`
- `A_{call+1}`
- `B_{ret}`

NB: Need to ensure bit patterns for nops appear nowhere else in code memory
Control-Flow Integrity

More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int $\rightarrow$ int function.

CFG excerpt

- $A_{call} \rightarrow B_1$
- $A_{call} \rightarrow C_1$
- $\text{succ}(A_{call}) = \{B_1, C_1\}$

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction.

1. Call $fp$
2. Check if $fp$ is not a nop IMM
3. If not, halt

$F_A$

$F_B$

$F_C$
**Control-Flow Integrity**

**Imprecise Return Information**

**Q:** What if $F_B$ can return to many functions?

**A:** Imprecise CFG

**CFG excerpt**

$$
\text{call } F_B \\
\text{nop IMM}_2 \\
\text{call } F_B \\
\text{nop IMM}_2 \\
\text{return}
$$

$$
\text{if}(**\text{esp} \neq \text{nop IMM}_2) \text{ halt}
$$

$$
\text{succ}(B_{\text{ret}}) = \{A_{\text{call}+1}, D_{\text{call}+1}\}
$$

**CFG Integrity:** Changes to the PC are only to valid successor PCs, per `succ()`.
Destination Equivalence

- Eliminate impossible return targets
  - Two destinations are said to be equivalent if connect to a common source in the CFG.

Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect call while a solid line stands for a direct call)
• Eliminate impossible return targets
  ‣ Can R2 be a return target of function_j?

Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect call while a solid line stands for a direct call)
Control-Flow Integrity

No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

A_{call} \rightarrow B_1 \rightarrow C_1

E_{call} \rightarrow C_1

CFG excerpt

A_{call} \rightarrow B_1 \rightarrow C_{1A}

E_{call} \rightarrow C_{1E}
Restricted Pointer Indexing

- One table for call and return for each function

- Why can’t function_j return to R2 with this approach?
Other Problems with CFI

- CFI enforcement can be expensive
- Idea: only check CFI for the last N branches
  - **kBouncer** inspects the last 16 indirect branches taken each time the program invokes a system call
    - Why 16? Uses Intel’s Last Branch Record (LBR), which can store 16 records
  - **ROPecker** also checks forward for future gadget sequences (short sequences ending in indirection)
- These hacks can be circumvented by extending the ROP chains
  - **Bottom line** – no shortcuts
Control-Flow Graph

- CFI enforces an expected CFG
  - Each call-site transfers to expected instruction
  - Each return transfers back to expected call-site
- Direct calls
  - Call instructions targeted for specific instruction – no problem
- Indirect calls
  - Function pointers – what are the possible targets?
- Returns
  - Determine return target dynamically – can be overwritten
- For the CFG, we need to compute the possible targets
Control-Flow Graph

- Computing an accurate estimate of the indirect call and return targets is intractable in general
  - Depends on predicting the value of a pointer
  - I.e., solving the points-to problem (undecidable)
- OK, maybe this is hard for function pointers (indirect calls), but this should be easy for returns, right?
  - You return to one of the possible callers
Control-Flow Graph

- Computing an accurate estimate of the indirect call and return targets is intractable in general
  - Depends on predicting the value of a pointer
  - I.e., solving the points-to problem (undecidable)

- OK, maybe this is hard for function pointers (indirect calls), but this should be easy for returns, right?
  - You return to one of the possible callers
  - Actually, there are exceptional cases – more later
Forward Edges

- How do we compute the possible targets for function pointers?
Forward Edges

- How do we compute the possible targets for function pointers?
- What are the possible legal targets of function pointers (i.e., indirect call sites)?
Forward Edges

- How do we compute the possible targets for function pointers?
- What are the possible legal targets of function pointers (i.e., indirect call sites)?
  - Any function
  - Called *coarse-grained CFI*
  - As this is the maximal set of legal function pointer targets, it is coarse
Coarse-grained CFI

```c
void (*fp1)()

void (*fp2)()
```
Forward Edges

- How do we compute the possible targets for function pointers?

- What are the possible legal targets of function pointers (i.e., indirect call sites)?
  - Any function
  - Called coarse-grained CFI
  - As this is the maximal set of legal function pointer targets, it is coarse

- This approach was applied by researchers – and then broken (easily) by other researchers
  - What are some options that would be more accurate?
Expected Targets

• How do we compute the possible targets for function pointers?

• What are the expected targets of an indirect call?
Signature-based CFI

- How do we compute the possible targets for function pointers?
- What are the expected targets of an indirect call?
  - Functions with the same type signature as the function pointer
  - Suppose you have a function pointer “int (*fn)(char *b, int n)”
    - Which functions should be assigned to that function pointer?
Signature-based CFI

• How do we compute the possible targets for function pointers?

• What are the expected targets of an indirect call?
  ‣ Functions with the same type signature as the function pointer
  ‣ Suppose you have a function pointer “int (*fn)(char *b, int n)”
    • Which functions should be assigned to that function pointer?

• Compute the set of functions that share that signature assuming any of these can be a target
  ‣ Fewer than all functions
  ‣ Intuitively seems like an overapproximation
  ‣ Can a function “void foo(void)” be assigned to the fptr above?
Intuitively, we can track the flow of function pointers from assignment to use.
Taint-based CFI

• For each function that may be assigned to a function pointer (e.g., foo)
  ‣ We taint the variables assigned to propagate to find which are used at which indirect call site

• How we propagate the taint depends on the type of LHS
  ‣ function pointer variable
  ‣ function pointer element in an array
  ‣ function pointer field in a structure
Taint Analysis

&foo → fptr

&foo

fptr
Taint Analysis

&foo

array[i]
Taint Analysis

&foo

struct op

.open
.read
.write

.open
.read
.write

.open
.read
.write

.open
.read
.write

.open
.read
.write
Assumptions

1. No arithmetic operations on function pointers

   void (*fptr)(int) = &foo;
   fptr += 10;

2. No data pointers to function pointers

3. No type casts from data pointer types (int *) to function pointer types
Example: FreeBSD

The average number of targets per indirect branch

- Coarse-grained CFI [CFI CCS’05]: 140K
- Signature-based CFI [MCFI PLDI’14]: 36
- Taint-based CFI [IEEE Euro S&P’16]: 10
Distribution of Taint Targets

Distribution of the number of targets for indirect branches

9.42 targets per indirect branch
(>3X reduction via signature-based)
Returns

• What to do about all the returns?
  ‣ Almost takes use back to the “destination equivalence” or “zig zag” problem
  • What was that again?
Returns

- What to do about all the returns?
  - Almost takes use back to the “destination equivalence” or “zig zag” problem
    - What was that again?

- But, we know that returns typically follow calls
  - How can we utilize that?
Shadow Stack

- Method for maintaining return targets for each function call reliably

- On call
  - Push return address on the regular stack
  - Also, push the return address on the shadow stack

- On return
  - Validate the return address on the regular stack with the return address on the shadow stack

- Why might this work? Normal program code cannot modify the shadow stack memory directly
Shadow Stack

- Intel Control-Flow Enforcement Technology (CET)
  - Has been announced
  - Not in products yet
- Goal is to enforce shadow stack in hardware
  - Throw an exception when a return does not correspond to a call site
- Challenge: Exceptions
  - There are cases where call-return does not match
  - E.g., Tail calls, thread libraries (setjmp, longjmp)
Take Away

- **Memory errors** are the classic vulnerabilities in C programs (**buffer overflow**)
- Need two steps to exploit memory errors
  - **Illegal memory write** – often, but not always, initiated by overflow
  - **Direct control flow** – to adversary-chosen code
- Defenses have been proposed to prevent both steps
  - **Bounds checks** via bounds metadata and/or fat pointers
  - **Control-flow integrity** has been suggested as the way to block ROP attacks