Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP

Control-flow Integrity (CFI)
Attacks against CFI and more defenses
Attacker Modus Operandi

Find memory corruption bug
  - Manipulate to take over program counter

Find ASLR bypass
  - Leak memory layout
  - Spray memory
  - Weakly or non-randomized sections/memory

Inject ROP payload
  - Break W^X semantics

Inject code
Find memory corruption bug
  - Manipulate to take over program counter

Control-flow Integrity aims to restrict the arbitrary manipulation of the program counter
Control-Flow Hijacking Prone Statements

Statements where the target statement cannot be known a priori

- Indirect control-flow transfers

Indirect calls, returns, and some switches

Calls to virtual functions are indirect calls

```c
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);

switch (cond) {
  val1: ... break;
  val2: ... break;
}

Class C {
  virtual void vcall(void);
}
C obj = new C();
obj->vcall();
```
Easily Observable in Machine Code

C Code

```
return;
return 100;
```

```
switch (cond) {
  val1: ... break;
  val2: ... break;
}
```

```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

```
Class C {
  virtual void vcall(void);
}
```

```
C obj = new C();
obj->vcall();
```

Machine Code

```
ret
```

```
jmp *(%rax)
```

```
jmp *(%rax)
call *(%rax)
```

```
call *(%rax)
```
Non-fixed Pointer Arguments

Indirect branch instruction

- `ret`
- `jmp *%rax`
- `call *%rax`

Pointer location

- `(%rsp)`
- `%rax`
- `(%rax)`

Stevens Institute of Technology
Non-fixed Pointer Arguments

Indirect branch instruction

\[ \text{ret} \]

\[ \text{jmp *%rax} \]
\[ \text{call *%rax} \]

Pointer location

\[ (%rsp) \]

\[ %rax \]

\[ (%rax) \]

CFI aims to restrict what these instructions can target

How?
A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. --wikipedia

Nodes are basic blocks (bbl)
Basic Blocks

In this case a bbl is a sequence of instructions with a single entry and single exit

Execution can enter the bbl at the first instruction

Execution can leave the bbl at the last instruction

Note: asynchronous events (e.g., signal) can temporarily transfer control flow elsewhere
CFG Example

call

call *

je

jg

ret

ret
CFG Example

call

je

ret

jg

ret

call *

Stevens Institute of Technology

Fall 2018
Extracting the CFG

With source code

- More reliable
- Cannot be fully reconstructed
- Resolving pointers is hard

```c
static void (*fptr)(char *string, int len);
void set_callback(void *ptr)
{
    fptr = ptr;
}
void process_items()
{
    for (string *s : items) {
        fptr(s->c_str, s->len);
    }
}
```

**Pointer aliasing.** In computer programming, *aliasing* refers to the situation where the same memory location can be accessed using different names. For instance, if a function takes two pointers A and B which have the same value, then the name A[0] aliases the name B[0].
Extracting the CFG

**With source code**
- More reliable
- Cannot be fully reconstructed
- Resolving pointers is hard

**Without source code**
- Requires accurate disassembly
- Cannot accurately define all paths
- Shared libraries are easier to handle

```c
static void (*fptr)(char *string, int len);

void set_callback(void *ptr)
{
    fptr = ptr;
}

void process_items()
{
    for (string *s : items) {
        fptr(s->c_str, s->len);
    }
}
```
Working with an Imperfect CFG

Let's assume that we know/can learn

- The location of every function
- The location of every indirect branch instruction

Coarse-grained CFI can enforce the following

- Indirect calls should only transfer control to functions
  - Same for most jumps
- Returns should only transfer control to instructions following an indirect call or jump
- More permissive than the actual (potentially unknown) CFG but better than before
What is Allowed

Indirect calls should only transfer control to functions

call *(%rax)

OK

Function_A:
ret

OK

Function_B:
ret
What is Allowed

Returns should only transfer control to instructions following a indirect call or jump

call *(%rax)
ret

Function_A:
ret

Function_B:
ret
What is Not Allowed

Indirect calls/jumps cannot target non function entry points

- But can target functions that could be called through an indirect call

Function_A:
ret

Function_B:
ret

Function_C:
pop %rax
ret

NOT
OK

OK
What is Not Allowed

Returns cannot target bytes not following a call/jump

- But can target valid bytes in functions that may have not called them

Function_A:
```
call *(%rax)
ret
```

Function_B:
```
call *(%rax)
pop %rax
ret
```

OK

NOT OK
ID codes are embedded into the binary program to identify acceptable targets

- 2-ID policy

Function_A:

```
[ID_1]
```

ret

Function_B:

```
[ID_1]
```

ret
Enforcing Through Embedded IDs

Checks are introduced right before the control transfer

Function_A:

```
check [ID_1]
check [ID_2]
ret
```

Function_B:

```
check [ID_1]
check [ID_2]
ret
```

This is not an instruction
Modifications for CFI Enforcement

**Function_A**:  
* (%rax) == ID_1  
call *(%rax+8)

**Function_B**:  
ret  
check [ID_2]

prefetchnta *(0xAABBCCDD)

3E 0F 18 05 DD CC BB AA
Modifications for CFI Enforcement

Function_A:

* (%rax) == ID_1
call *(%rax+8)

check [ID_1]
call *(%rax)

Function_B:

ret

check [ID_2]
ret
call *(%rax)

Function_B:

ret

check [ID_2]
ret

pop %rcx
*(%rcx+4) == ID_2
jmp *(%rcx)

This instruction does not have an adverse effects

prefetchnta *(0xAABBCCDD)

3E 0F 18 05 DD CC BB AA

Stevens Institute of Technology
Control-flow integrity

Martín Abadi  University of California, Santa Cruz and Microsoft Research, Santa Cruz, CA
Mihai Budiu  Microsoft Research
Úlfar Erlingsson  Reykjavík University and Microsoft Research
Jay Ligatti  University of South Florida, Tampa, FL

ACM Transactions on Information and System Security (TISSEC)

http://dl.acm.org/citation.cfm?id=1609960

Limitations:
• Code integrity must be ensured (no code injection)
• Incremental deployment is not supported (all or nothing)
• Only 2 IDs are supported for enforcing CFI
Practical Control Flow Integrity and Randomization for Binary Executables

Chao Zhang
Tao Wei
Zhaofeng Chen
Lei Duan
Laszlo Szekeres
Stephen McCamant
Dawn Song
Wei Zou

Proceedings of the 2013 IEEE Symposium on Security and Privacy

http://dl.acm.org/citation.cfm?id=2498134
Three IDs are used to restrict control flow

Function_A:

check [ID_1]
ret

call *(%rax)

Sensitive_Function_A

call 0x...

check [ID_2 | ID_3]
ret
Three IDs are used to restrict control flow

**Function_A:**

```
check [ID_1]
call *(%rax)
check [ID_2]
ret
```

**Sensitive_Function_A**

```
call 0x....
check [ID_2 | ID_3]
ret
```

Memory allocation routines, changing permissions, launching processes, etc.
Three IDs are used to restrict control flow

Function_A:
- check [ID_2]
- ret

Sensitive_Function_A
- call 0x...
- check [ID_2 | ID_3]
- ret

CCFIR

Stevens Institute of Technology

Fall 2018
Three IDs are used to restrict control flow

Function_A:
- check [ID_2]
- ret

Sensitive_Function_A:
- call 0x...
- check [ID_2 | ID_3]
- ret

Prevents code-reuse of sensitive functions
Sensitive Functions Heuristic

Function_A:

check [ID_2]
ret

Prevents code-reuse of sensitive function parts

Sensitive_Function_A:
call 0x...

Sensitive_Function_B:
check [ID_2 | ID_3]
ret

Stevens Institute of Technology
Each indirect call is redirected through a trampoline using a direct jump.

Targeted functions are called indirectly through another trampoline.

Direct control transfer

Indirect control transfer

\[ M_F = 0x8000007 \]
\[ M_R = 0x800000f \]
\[ \text{or} \]
\[ M_R = 0xC00000f \]
Any other return instruction’s target must be a valid return stub (i.e., 16-byte aligned). Indirect call/jump instructions’ targets must be function entries. The target of a return back to a sensitive function can be any valid return stub (i.e. 16-byte aligned). For any indirect call/jump instruction, its target should be 0x800000f for returns from functions called by sensitive instructions, and 0xc00000f for all other return instructions. The target address is bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the mask 0x8000007 (i.e. bit 0). If the result is non-zero, the control flow is directed to a predefined error handler (i.e. the 3rd bit is 1). Thus if the target address is bitwise ANDed with 8, the result should be non-zero. In addition, if the TARGET is bitwise ANDed with 8, the result should be aligned (i.e. the 0-2 bits are 0, but the 3rd bit is 1). Thus if the byte aligned (i.e. the 27th bit is 0) and only 8-bytes are performed using one or two bit-testing instructions.

As discussed in Section III-D, this enforcement can be carefully to perform checks. Function stubs are not restricted to be within the current module’s Springboard. But the stubs are always forced to the jump table entries and thus cannot be altered. While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect jump instruction, to improve performance. Hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance. In this way, the control flow is forced to one entry in its jump table. For example, GCC binary generated by modern compilers (e.g., GCC and VC) need dynamic checks. For any switch statement, regardless of what its control expression is, the control flow in the first makes a bound check against the bound value. If it exceeds the bound, then the control flow transfers to the default case value in switch statements). If it exceeds the bound, then the control flow transfers to the default case (corresponding to the default case).

The function stubs are also permitted, since their addresses are compatible; they can be any valid return stub (i.e. 16-byte aligned). In practice (e.g. system DLLs on Windows 7 cannot be longjmp’d). While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect jump instruction, to improve performance. Hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance. In this way, the control flow is forced to one entry in its jump table. For example, GCC binary generated by modern compilers (e.g., GCC and VC) need dynamic checks. For any switch statement, regardless of what its control expression is, the control flow in the first makes a bound check against the bound value. If it exceeds the bound, then the control flow transfers to the default case value in switch statements). If it exceeds the bound, then the control flow transfers to the default case (corresponding to the default case).

A protected module only allows indirect control transfers to a Springboard stub (i.e., 16-byte aligned with the 26th bit set). The target of a return back to a sensitive function can be any valid return stub (i.e. 16-byte aligned). For any indirect call/jump instruction, its target should be 0x800000f for returns from functions called by sensitive instructions, and 0xc00000f for all other return instructions. The target address is bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the mask 0x8000007 (i.e. bit 0). If the result is non-zero, the control flow is directed to a predefined error handler (i.e. the 3rd bit is 1). Thus if the target address is bitwise ANDed with 8, the result should be aligned (i.e. the 27th bit is 0) and only 8-bytes are performed using one or two bit-testing instructions.

As discussed in Section III-D, this enforcement can be carefully to perform checks. Function stubs are not restricted to be within the current module’s Springboard. But the stubs are always forced to the jump table entries and thus cannot be altered. While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect jump instruction, to improve performance. Hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance. In this way, the control flow is forced to one entry in its jump table. For example, GCC binary generated by modern compilers (e.g., GCC and VC) need dynamic checks. For any switch statement, regardless of what its control expression is, the control flow in the first makes a bound check against the bound value. If it exceeds the bound, then the control flow transfers to the default case value in switch statements). If it exceeds the bound, then the control flow transfers to the default case (corresponding to the default case).
Function stub address
\[
\text{AND} \\
M_F = 0x8000007
\]
Section IV-C2.

R5.

R4.

we find only one case of binaries, although it's more popular in malicious code. In our experiments permitted to flow to an aligned address in the Springboard section.

sections, and all indirect control transfers (dashed lines) are only 

SFI \[38\][39\] to make CFI enforcement more efficient.

we conversely use ideas of layout-based checking from make software fault isolation (SFI) more efficient \[13\],

D. The Springboard and New Memory Layout

targets are used (i.e., all control-transfer instructions).

function pointers and return addresses) and where transfer 

where transfer targets are created (i.e., all occurrences of

for large binaries, and can be easily reviewed. These remaining parts are usually small even binaries not respecting R5, we can still identify most code 

plying with rules R1

we take an approach that can disassemble a PE file com-

disassembled recursively to identify all possible instructions.

indirect code entries. Then with the help of the export 

1

While CFI enforcement techniques have been used to 

As binaries can be disassembled correctly, we can identify 

getpc() 

Combined with other policies described in Section IV-B, 

These rules hold for binaries generated by modern com-

Original code section

EntryPoint

4

3

2

1

Function stub address

AND

M_F = 0x8000007

128MB segments

8-byte aligned slots

0

Compiler separate code and data sections (in order to 

conform to DEP). In code sections, the only data which 

can be indexed through relocation tables (as rule R1).

languages.

Figure 2: Memory layout for executables hardened

with this rule, and even most inline assembly code does.

Real-world applications' code sections are typically small-

For each module, we introduce a new code section called 

Make Valid Targets Distinguishable.

First, function pointer stubs and return address stubs are 

Discussed in Fall 2018

Second, return address stubs for return-landing 

stubs within the Springboard are further distinguishable.

to defeat advanced attacks like ROP and return-to-libc, code 

are all redirected to code stubs in the Springboard. Further, 

distinguish valid targets of indirect transfer instructions from 

node 

control-transfer target (e.g. nodes 

Figure), the Springboard contains an associated unique stub 

5 

2’

3’

2’

5

4

3

2

1

(odd 128MB memory slice, as long as the whole section 

slices. Data sections are not constrained.

are always in even slices, and other code sections are in odd 

into 128MB-

ure 2, it is enforced that any executable code section whose 

memory areas through the memory layout. As shown in Fig-

arbitrary target becomes impossible. 

a result, diverting the execution to an attacker-supplied 

Make sure that 

the Springboard contains an associated unique stub 

M_F = 0x8000007

AND

0x6800 0000-0x6fff ffff (27bit is 1)

0x6000 0000-0x67ff ffff (27bit is 0)

0x3800 0000-0x3fff ffff (27bit is 1)

0x3000 0000-0x37ff ffff (27bit is 0)

0x6800 0000-0x6fff ffff (27bit is 1)

0x6000 0000-0x67ff ffff (27bit is 0)

0x3800 0000-0x3fff ffff (27bit is 1)

0x3000 0000-0x37ff ffff (27bit is 0)

128MB

128MB

128MB

128MB
Hoists check conditions before the control transfers. The policy our scheme enforces is the following:

- For any indirect call/jump instruction, its target should be a function's address (i.e., 16-byte aligned).
- Any other return instruction's target must be a valid return stub (i.e., 16-byte aligned).
- Indirect call/jump instructions' targets must be function addresses.
- The target of a return back to a sensitive function can be altered. Thus BitRewrite skips validating these instructions.

A protected module only allows indirect control transfers to a predefined error handler (i.e., an exceptional case).

In Figure 6, these bitwise AND operations are performed with the mask 0x8000007 (i.e., the 27th bit is 0) and only 8-bytes aligned. The target address is bitwise ANDed with 8, the result should be zero. If the result is non-zero, the control flow is directed to the default case. And then, the control flow transfers to the error handler.

The EIP value and the invalid transfer target are logged when the error handler is called. In our implementation, the error handler will log the buggy instruction. (To record the EIP, there is a separate copy of the EIP value for each indirect call/jump and return.)

The function in Figure 6, the result of the bitwise AND instruction is on the top of the stack, pointed to by the esp register. The mask 0x800000f (i.e., the 27th bit is 0) and only 8-bytes aligned. The return address is then tested against a mask 0x800000f for returns from functions called by sensitive functions, and 0xc00000f for all other return instructions.

However, rewriting all modules is not always possible in practice (e.g., system DLLs on Windows 7 cannot be altered). While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from a protected module to an unprotected one, the separate modules will be compatible with each other in the Springboard. In this case, the targets of the indirect control transfers from the protected module to the unprotected module are validated the same way. And thus if every module in a program (i.e., the main program and all DLLs) is rewritten, any combination and the control-flow integrity is enforced.

As discussed in Section III-D, this enforcement can be executed. Stubs within other modules' Springboard sections are not restricted to be within the current module's Springboard section. Stubs within other modules' Springboard sections whose targets are valid Springboard stubs. But the stubs are altered. Thus BitRewrite skips validating these indirect jump instructions, to improve performance.

The function in Figure 6, the result of the bitwise AND instruction is on the top of the stack, pointed to by the esp register. The mask 0x800000f (i.e., the 27th bit is 0) and only 8-bytes aligned. The return address is then tested against a mask 0x800000f for returns from functions called by sensitive functions, and 0xc00000f for all other return instructions.

However, rewriting all modules is not always possible in practice (e.g., system DLLs on Windows 7 cannot be altered). While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from a protected module to an unprotected one, the separate modules will be compatible with each other in the Springboard. In this case, the targets of the indirect control transfers from the protected module to the unprotected module are validated the same way. And thus if every module in a program (i.e., the main program and all DLLs) is rewritten, any combination and the control-flow integrity is enforced.

As discussed in Section III-D, this enforcement can be executed. Stubs within other modules' Springboard sections are not restricted to be within the current module's Springboard section. Stubs within other modules' Springboard sections whose targets are valid Springboard stubs. But the stubs are altered. Thus BitRewrite skips validating these indirect jump instructions, to improve performance.
Return stubs are also aligned

Original

Hardened

mov ecx, foo
...  
call ecx  
back:  
...

foo:
...

ret

mov ecx, foo_sb
...  
test ecx, 8  
jz error  
test ecx, M_F  
jnz error  
jmp back_sb-2  
back:  
...

foo:
...

test [esp], M_R  
jnz error  
ret

foo_sb:
  jmp foo

call ecx
back_sb:
  jmp back

Direct control transfer
Indirect control transfer

M_F = 0x8000007  
M_R = 0x800000f  
or  
M_R = 0xC00000f

Any other return instruction's target must be a valid return stub (i.e. 16-byte aligned).

The target of a return back to a sensitive function can be altered. While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, the control flow is hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance.

Figure 6: Rewriting of an indirect call and return

Indirect call/jump instructions' targets must be function pointers (i.e. 8-byte aligned but not 16-byte aligned), since we do not want to restrict the addresses of these functions. As discussed in Section III-D, this enforcement can be implemented in the Springboard (i.e. the 27th bit is 0) and only 8-bytes per indirect call/jump and return are validated the same way. And thus if every module in a program (i.e. the main program and all DLLs) is rewritten, program instructions' targets before the control transfers. The policy our scheme enforces is the following:

- Any other return instruction's target must be a valid return stub (i.e. 16-byte aligned).
- Indirect call/jump instructions' targets must be function pointers (i.e. 8-byte aligned but not 16-byte aligned).
- The target of a return back to a sensitive function can be altered. While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, the control flow is hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance.

Figure 6: Rewriting of an indirect call and return

Any Exceptional Case.

Figure 6 also shows how the validation is inserted before any valid return stub (i.e. 16-byte aligned).
Return stubs are also aligned

Return stub address
AND
M_R = 0x800000f

16-byte aligned slots

Return stubs are also aligned

Original

Hardened

mov ecx,foo
...
call ecx
back:
...

mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...

call ecx
back_sb:
jmp back

M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

Indirect control transfer
Direct control transfer

Figure 7: Rewriting of a direct call and return

mov ecx,foo
...
call ecx
back:
...

mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...

call ecx
back_sb:
jmp back

M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

Indirect control transfer
Direct control transfer

Figure 6: Rewriting of an indirect call and return

Figure 6 also shows how the validation is inserted before
for each indirect call/jump and return.)

Hardened

Original

16-byte aligned slots

Return stub address
AND
M_R = 0x800000f

16-byte aligned slots

Return stubs are also aligned

Original

Hardened

mov ecx,foo
...
call ecx
back:
...

mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...

call ecx
back_sb:
jmp back

M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

Indirect control transfer
Direct control transfer

Figure 7: Rewriting of a direct call and return

Figure 7 also shows how the validation is inserted before
for each indirect call/jump and return.)

Hardened

Original

mov ecx,foo
...
call ecx
back:
...

mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...

call ecx
back_sb:
jmp back

M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

Indirect control transfer
Direct control transfer
Direct calls to functions also go through trampolines but no checks required
D. Compatibility Issues

For any indirect call/jump instruction, its target should be a valid Springboard stub (i.e., 16-byte aligned with the 26th bit is 1). While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, it is altered. While control transfers from an unprotected module to a sensitive function can be altered. Thus BitRewrite skips validating these instructions' targets before the control transfers. The policy our scheme enforces is the following:

1. Indirect call/jump instructions' targets must be function pointers stubs (i.e. 8-byte aligned but not 16-byte aligned).
2. Return instructions' targets must be valid return address references.
3. Any other return instruction's target must be a valid return address.

The target of a return back to a sensitive function can be any 8-byte address. The target of an indirect jump can be any 16-byte address, while the target of an indirect call can be any 8-byte address. These targets are bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the EIP value and the invalid transfer target, and then terminate the process. (To record the EIP, there is a separate copy of the EIP value and the invalid transfer target, and then terminate the process.)

The function $\text{foo}$ has a 16-byte return stub address $0x800000f$ for returns from functions called by sensitive functions. The return address is then tested against a mask $0x8000007$ (i.e. the 27th bit is 0) and only 8-bytes are loaded from the 16-byte return address (corresponding to the 8 most significant bits). In this way, the control flow is directed to the default case. And then, the control flow transfers to the 8-byte address.

However, rewriting all modules is not always possible (e.g. system DLLs on Windows 7 cannot be modified). While control transfers from an unprotected module to an unprotected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, it is altered. While control transfers from an unprotected module to a sensitive function can be altered. Thus BitRewrite skips validating these instructions' targets before the control transfers. The policy our scheme enforces is the following:

1. Indirect call/jump instructions' targets must be function pointers stubs (i.e. 8-byte aligned but not 16-byte aligned).
2. Return instructions' targets must be valid return address references.
3. Any other return instruction's target must be a valid return address.

The target of a return back to a sensitive function can be any 8-byte address. The target of an indirect jump can be any 16-byte address, while the target of an indirect call can be any 8-byte address. These targets are bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the EIP value and the invalid transfer target, and then terminate the process. (To record the EIP, there is a separate copy of the EIP value and the invalid transfer target, and then terminate the process.)

The function $\text{foo}$ has a 16-byte return stub address $0x800000f$ for returns from functions called by sensitive functions. The return address is then tested against a mask $0x8000007$ (i.e. the 27th bit is 0) and only 8-bytes are loaded from the 16-byte return address (corresponding to the 8 most significant bits). In this way, the control flow is directed to the default case. And then, the control flow transfers to the 8-byte address.

However, rewriting all modules is not always possible (e.g. system DLLs on Windows 7 cannot be modified). While control transfers from an unprotected module to an unprotected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, it is altered. While control transfers from an unprotected module to a sensitive function can be altered. Thus BitRewrite skips validating these instructions' targets before the control transfers. The policy our scheme enforces is the following:

1. Indirect call/jump instructions' targets must be function pointers stubs (i.e. 8-byte aligned but not 16-byte aligned).
2. Return instructions' targets must be valid return address references.
3. Any other return instruction's target must be a valid return address.

The target of a return back to a sensitive function can be any 8-byte address. The target of an indirect jump can be any 16-byte address, while the target of an indirect call can be any 8-byte address. These targets are bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the EIP value and the invalid transfer target, and then terminate the process. (To record the EIP, there is a separate copy of the EIP value and the invalid transfer target, and then terminate the process.)

The function $\text{foo}$ has a 16-byte return stub address $0x800000f$ for returns from functions called by sensitive functions. The return address is then tested against a mask $0x8000007$ (i.e. the 27th bit is 0) and only 8-bytes are loaded from the 16-byte return address (corresponding to the 8 most significant bits). In this way, the control flow is directed to the default case. And then, the control flow transfers to the 8-byte address.

However, rewriting all modules is not always possible (e.g. system DLLs on Windows 7 cannot be modified). While control transfers from an unprotected module to an unprotected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected one, it is altered. While control transfers from an unprotected module to a sensitive function can be altered. Thus BitRewrite skips validating these instructions' targets before the control transfers. The policy our scheme enforces is the following:

1. Indirect call/jump instructions' targets must be function pointers stubs (i.e. 8-byte aligned but not 16-byte aligned).
2. Return instructions' targets must be valid return address references.
3. Any other return instruction's target must be a valid return address.

The target of a return back to a sensitive function can be any 8-byte address. The target of an indirect jump can be any 16-byte address, while the target of an indirect call can be any 8-byte address. These targets are bitwise ANDed with 8, the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the EIP value and the invalid transfer target, and then terminate the process. (To record the EIP, there is a separate copy of the EIP value and the invalid transfer target, and then terminate the process.)

The function $\text{foo}$ has a 16-byte return stub address $0x800000f$ for returns from functions called by sensitive functions. The return address is then tested against a mask $0x8000007$ (i.e. the 27th bit is 0) and only 8-bytes are loaded from the 16-byte return address (corresponding to the 8 most significant bits). In this way, the control flow is directed to the default case. And then, the control flow transfers to the 8-byte address.
Microsoft’s Control-Flow Guard

Included in MS Visual Studio

Inserts control-flow checks before indirect calls during compilation

A bitmap marks the allowed targets

check bitmap[\%rax]
call *(\%rax)

bitmap:

1 bit per 8 or 16-byte slot

Compiled with CFG
Microsoft’s Control-Flow Guard

Included in MS Visual Studio

Inserts control-flow checks before indirect calls during compilation

A bitmap marks the allowed targets

check bitmap[%rax]
call *(%rax)

bitmap:

1 bit per 8 or 16-byte slot
Topics

Attackers shift towards client programs
Back to return-to-libc
Return-oriented programming
Fine-grained code randomization
JIT-ROP
Control-flow Integrity (CFI)
Attacks against CFI and more defenses
Reachable Targets Under CFI

Most instructions cannot be targeted (> 98%)
What is Left

Call Sites (CS)

- Targetable by `return` instructions
- CS gadgets
- Return Oriented Programming (ROP)

Function Entry Points (EP)

- Targetable by `indirect call` and `indirect jump` instructions
- EP gadgets
- Call Oriented Programming (COP)
CS gadgets: Linking

call ... CS
ret

call ... CS
ret

call ... CS
ret
CS gadgets: Linking

call ...
ret

call ...
ret

call ...
ret

stack

gadget address
gadget address
gadget address

Fall 2018
Stevens Institute of Technology
CS gadgets: Linking
CS gadgets: Calling Functions

```
Function X:
    ret
    # call ...
    # CS
    call *(%rsi)
    ret
    # call ...
    # CS
    ret
    # call ...
    # CS
    ret
    # call ...
    # CS
```

Stevens Institute of Technology
CS gadgets: Calling Sensitive Functions

CCFIR: No indirect calls to sensitive APIs

VirtualProtect:

Call ...
ret

Call ...
ret

call *(%rsi)
ret

call ...
ret

Stevens Institute of Technology
CS gadgets: Calling Sensitive Functions

VirtualProtect:

- call ...
- call *(%rsi)
- ret

- call ...
- call 788..
- ret

- call ...
- ret

- call ...
- ret
EP gadgets: Linking

Chaining is significantly harder

Function_X:
- call *(%rax)

Function_Y:
- call *(%rax)

Function_Z:
- call *(%rax)
EP gadgets: Calling Functions

Function_X:
```
call *(%rax)
```

Function_Q:
```
call *(%rbx)
call *(%rdx)
```

Function_Z:
```
call *(%rax)
```

memset:
```
call *(%rbx)
call *(%rdx)
```

ret
EP gadgets: Calling Functions

**Function_X:**
- `call *(%rax)`

**Function_L:**
- `call *(%rax)`
- `call *(%rdx)`

**Function_Q:**
- `call *(%rbx)`
- `call *(%rdx)`

**Function_Z:**
- `call *(%rax)`

**memset:**
- `call *(%rbx)`
- `call *(%rdx)`

**ret**
Switch Control: CS → EP

Function_X:
- call *(%rax)
- call *(%rax)
Switch Control: EP → CS

Function_X:
call *(%rax)

Function_Y:
call ...
ret

Need to corrupt return address

call ...
ret

Stevens Institute of Technology
Switch Control: EP → CS

Function_X:
- call *(%rax)

Function_Y:
- ret

Need to corrupt return address

Corrupt stack by
- breaking calling conventions
- Self-corrupting function (e.g., memcpy())
Compromising Coarse-grained CFI is Possible


Exploiting Internet Explorer 8

- Vulnerability: Heap Overflow (CVE-2012-1876)
- More info about vulnerability @ http://www.vupen.com/blog

Assume ASLR / DEP / CCFIR in place

First controlled indirect branch instruction: \texttt{jmp edx}\n(EP → CS) + VirtualProtect + memcpy = Code Injection
Finer-Grained CFI

Various approaches to improve CFI

- More accurate CFG and more checks
- Only allow calls to target the functions they actually were intended to
  - Better forward-edge CFI

Context-sensitive control flow enforcement

- For example, a function should return to its caller not any caller
Shadow Stacks

Regular stack

- return address
- saved rbp
- local variables
- return address
- saved rbp
- local variables

Shadow stack

- return address
- return address

This results into multiple instructions

call f
...

f:
ssp = 8
ssp = *sp
...
...
*ssp = *rsp
if NZ then error
ret
Shadow Stacks

This results into less instructions

\[
f: \quad *(sp+off) = *sp \\
\ldots \\
\ldots \\
*(sp+off) == *sp \\
\text{if NZ then error ret}
\]
Shadow vs (Un)safe Stacks

Unsafe stack
- saved rbp
- local variables

Safe stack
- return address
- return address

A separate register (not %rsp) used
Shadow Stack Limitations

Performance is the main obstacle for adoption

- The Performance Cost of Shadow Stacks and Stack Canaries
- [https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf](https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf)

Intel announced that hardware support for shadow stacks and CFI (called control-flow enforcement) will be made available on their future CPUs

- [http://www.theregister.co.uk/2016/06/10/intel_control_flow_enforcement/](http://www.theregister.co.uk/2016/06/10/intel_control_flow_enforcement/)
Heuristics-based Approaches

kBouncer: Efficient and Transparent ROP Mitigation

- Vassilis Pappas et al. [Usenix Security ‘13]
- Winner of Microsoft’s Blue hat prize

Use HW debugging feature to detect abnormal control-flow transfers

- Low overhead!
**Last Branch Record (LBR)**

CPU registers store last branches taken by the program
- Ring-buffer structure

Holds last 16 entries
- Store source:destination

Configurable
- Example: Store only indirect calls
Detection Approach

1. Returns must target call sites

2. A limited number of small code fragments can be chained together

Max gadget size

Max chain length
Fast Checks

The payload will eventually interact with the OS through system calls

- Check for abnormal control transfers on system call entry
Detection Approach

1. Returns must target call sites

2. A limited number of small code fragments can be chained together

How can we establish the max gadget size and max chain length?

Max gadget size

Max chain length
Establishing The Parameters

Set max gadget size to 19 (<20)

Evaluate max chain length experimentally

Dataset: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office (Word, Excel, Powerpoint)
# Chosen Parameters

<table>
<thead>
<tr>
<th></th>
<th>kBouncer</th>
<th>ROPecker</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time-of-Check</strong></td>
<td>Entry of Sensitive API</td>
<td>Entry of Sensitive API + during execution</td>
</tr>
<tr>
<td><strong>Gadget Length</strong></td>
<td>20 instructions</td>
<td>6 instructions</td>
</tr>
<tr>
<td><strong>Inspect BH instances</strong></td>
<td>Detected max &quot;benign&quot; gadget chain length: 5</td>
<td>Detected max &quot;benign&quot; gadget chain length: 10</td>
</tr>
<tr>
<td><strong>Gadget Chain Length</strong></td>
<td>8 gadgets</td>
<td>11 gadgets</td>
</tr>
</tbody>
</table>

Approach similar to kBouncer
Why Picking Parameters Is Hard

Executing a legitimate program

Max gadget size

Max chain length

Security Check

No alert, all is good!
Why Picking Parameters Is Hard

Executing a legitimate program

Max gadget size

Max chain length

Security Check

False positive!
Why Picking Parameters Is Hard

Executing a legitimate program

Max gadget size

Max chain length

Security Check

False positive!
How to Avoid Detection?

Interpose longer gadgets in the exploit

Max gadget size

Max chain length

No alert, all is good!
Using Long Gadgets

Long gadgets frequently:

- Use a high number of registers
- Leave used registers dirty at exit
- Require memory preparations to avoid crashing
- Have whacky code sequences

```assembly
mov eax, ebx
mov ecx, edx
add esi, edi

mov esi, [0x1234]
cmp esi, 10
jg X

mov ecx, 0x2321
div ecx
mov [eax], edi

mov ecx, 0x5678
and edi, ecx
xor eax, edi
retn
```
Such Defenses Are Also Vulnerable

Exploiting Internet Explorer 8 similar to CFI attack

Assumes kBouncer is in place

- Also applies to similar defenses like ROPecker [NDSS ‘13]

Multiple payloads

- kBouncer thresholds: $T_C=6, T_G=20$
- Stricter thresholds: $T_C=2, T_G=27$
Per Application Thresholds

![Graph showing the number of instances vs. gadget-chain length for various applications. The graph includes data for Acrobat, IE (Google), IE (YouTube), Excel, Word, PowerPoint, and WMPlayer. The y-axis represents the number of instances on a logarithmic scale, ranging from 0.1 to 1000000, and the x-axis represents the gadget-chain length ranging from 0 to 5.]
What if We Had the Perfect CFG

We know exactly which functions are called from an indirect call

We know exactly the call sites where a function’s return is supposed to return

But we still do not have a shadow stack

Control Flow Bending

How to Exploit the `memcpy()` Hotspot

`memcpy(dst, src, N)`

Assume `memcpy` is not buggy

`some_function:`

```c
... ...
memcpy(dst, src, N)
... ...
```
How to Exploit the memcpy() Hotspot

memcpy(dst, src, N)
Dispatcher Function

memcpy() acts as a dispatcher function
  - Can be used to return to gadgets part of the CFG

Other hot functions can act as dispatcher functions, as long as:
  - They are commonly called
  - Their arguments are under attacker control
  - Can overwrite their own return address
Summary

CFI is a powerful security primitive

Depends on the quality/accuracy of the CFG

Even in the ideal case, it might fall to code-reuse attacks
  - Depends on the application
    - Complexity of the CFG
    - Availability of gadgets