A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level

Yunxi Yang

December 16th 2016
Enforce control-flow integrity without access to source code is hard in practice.

Most existing binary-level CFI implementations base their invariants on an approximation of the CFG which leaves enough wiggle room for an attacker.
TypeArmor

- binary-level protection prototype, can stop all COOP attacks published to date.
- provides strong mitigation for many types of code-reuse attacks for programs binaries, without requiring access to source code.
Introduction
Counterfeit Object-Oriented Programming (COOP)

Process()
{
    ...
    while(condition)
    {
        ...
        call fptr
        ...
    }
}

Func1()
{
    ...
}

Func2()
{
    ...
}

Func3()
{
    ...
}
Introduction
Data Flow in COOP

- Implicit data flow
  - data flow using unused argument registers

- Explicit data flow
  - data flow using overlapping counterfeit object fields or global variables
  - data flow by relying on arguments actually passed to the callee
TypeArmor: Invariants for Targets and Callsites

- **CFI:**
  - target-oriented invariants
  - based on traditional CFI policies
  - underestimation

- **CFC:**
  - callsite-oriented invariants
  - first time to explore in binary level
  - overestimation
TypeArmor: Invariants for Targets and Callsites

CFI

movl %rax, %rdi
movl %rbx, %rsi
mov 0x123, %rdx
call *0x8(%rax)

movl 0x44, %rdi
call *%rax

call with 3 arguments

call with 1 argument
TypeArmor: Invariants for Targets and Callsites

CFC

- instruments each indirect callsite to scramble unused arguments before transferring control to the callee
- instruments each void function to scramble unused return arguments before transferring control back to the caller
Overview
TypeArmor’s Impact on COOP

CFI:
- reduces the target set of the virtual function calls by the main-loop and recursive gadgets considerably
- prohibits any forward edges to functions that expect more arguments than the callsite prepares

CFC:
- enforces a maximum number of arguments prepared at a callsite and scrambles the unused registers
- destructively updates unused argument registers before an indirect call and thus mitigates data-passing
Callee Analysis
Forward Analysis

- Register states
  - read-before-write(R)
  - write-before-read(W)
  - clear/untouched(C)
Callee Analysis
Forward Analysis

- **Process**
  - If all argument registers are either R or W, the analysis terminates
  - If at least one register is C, a recursive forward analysis starts until the block has no outgoing edges
  - A recursive analysis loops over all outgoing edges of the basic block to get a pointer to the next basic block to analyze
edge types
- Direct calls
- Indirect calls
- Returns
- Others
Callee Analysis
Merging Paths

- $S_i (i = 1, 2, \ldots n)$
- The state for argument register $c$ in $S$ can only be $R$ if the state for $c$ is $R$ for all states $S_i (i = 1, 2, \ldots n)$
- Combine $S$ with $S_B$
- States other than $C$ in $S_B$ always supersede states in $S$
Callee Analysis
Argument Count

- $rdi, rsi, rdx, rcx, r8, r9$
- if($r9 == R$)
  - then the function expects at least 6 arguments
- else examine $r8$
- if($r8 == R$)
  - then the function expects at least 5 arguments
- else examine $r7$
- ...

...
Figure 3. Callee analysis for `void set_errno(address_item *addrlist, int errno_value, uschar *msg, int rc, BOOL pass_message)` (of Exim). Observe how merging paths works. The basic block starting at 0x47edf1b (emphasized in bold) has state $S_B = (R, C, R, C, R, C)$, since rdi, rdx, and r9 are read. There are two incoming states to this block, namely $S_1 = (R, C, R, C, R, C)$ and $S_2 = (R, C, R, C, R, C)$, which are combined to a superstate $S = (R, C, C, C, C, C)$ (note that $C$ always supersedes). Finally, the superstate is combined with the block state, but this time $R$ supersedes and hence the output state is $(R, C, C, C, C, C) \land (R, C, R, C, C, R, C, R, C, R) = (R, C, R, C, C, *). The final state of all analyzed blocks is $(R, R, C, C, C, *), where the * denotes that $C$ does not supersedes $W$ or vice versa.
Callsite Analysis
Backward Analysis

- Register states
  - set(S)
  - trashed(T)
Process
- If all argument registers are S, TypeArmor stops the analysis and assumes that the callsite uses the maximum number of arguments.
- If some arguments cannot be considered either S or T and the basic block has incoming edges, TyperArmor starts a recursive backward analysis.
Callsite Analysis
Backward Analysis

- edge types
  - Direct calls
  - Returns
  - Others
Callsite Analysis
Merging Paths

- T always supersedes S
- Once the recursive analysis is finished, the number of prepared arguments is set based on the states of the last write operations
Example of Callsite Analysis Operation

```c
#define RESPONSE_WRITE_NUM_STR(strm, fmt, numeric, msg)  
    pr_trace_msg(trace_channel, 1, (fmt), (numeric), (msg));  
if (resp_handler_cb)  
    pr_netio_printf((strm), "%s", resp_handler_cb(resp_pool,(fmt),  
        (numeric), (msg)));  
else  
    pr_netio_printf((strm), (fmt), (numeric), (msg));

void pr_response_flush(pr_response_t **head) {
    .......
    RESPONSE_WRITE_NUM_STR(session.c->outstrm, "%s %s\r\n",  
        last_numeric, resp->msg)
    .......
}
```

```
426d51: callq 434960 <pr_trace_msg>  |  426e65: mov 0x5ea4c(%rip),%rdi
426d56: mov 0x5eb73(%rip),%r8    |  426e6c: mov 0x8(%rbx),%rdx
426d5d: test %r8,%r8           |  426e70: mov 0x10(%rbx),%rcx
426d60: mov 0x67c79(%rip),%rax  |  426e74: mov $0x463a60,%esi
426d67: mov 0x38(%rax),%r15     |  426e79: xor %eax,%eax
426d6b: jne 426e65             |  426e7b: callq *%r8
```
Return Values

- A non-void callsite should never target a void callee
- A void callsite is allowed to target both void and non-void callees
Return Values

Non-void callsites
- detected by applying forward analysis from the start of a callsite

Void callees
- detected by applying backward analysis at the exit points of a function
Shadow Code Memory Preparation

- a copy of the original code that also contains the instrumentation of the callsites.
- Program execution is performed using the shadow code.
- Using shadow code, we do not have to ensure that label does not overwrite code.
CFI enforcement

**Callee instrumentation**
- use 3 bits to represent seven possible labels (no arguments(0) to all arguments(6))
- use 1 bit to represent whether the function returns a value

**Callsite instrumentation**
- retrieve the callee’s label
- decode the information
- check if the callee is compatible with the callsite
CFC enforcement

- scramble unused registers at indirect callsites
- essentially enforce a zero percent underestimation rate at the callee
Overview of How TypeArmor addresses CRAs

TypeArmor stops existing code-reuse exploits. Since TypeArmor specifically targets x86_64 binaries, the IE 32-bit COOP exploit is out of scope. Note that even without deploying CFC, TypeArmor stops all exploits.

<table>
<thead>
<tr>
<th>Exploit</th>
<th>Stopped?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COOP ML-G [26]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- IE (32-bit)</td>
<td>✗</td>
<td>Out of scope</td>
</tr>
<tr>
<td>- IE 1 (64-bit)</td>
<td>✓ (CFI)</td>
<td>Argcount mismatch</td>
</tr>
<tr>
<td>- IE 2 (64-bit)</td>
<td>✓ (CFI)</td>
<td>Argcount mismatch</td>
</tr>
<tr>
<td>- Firefox</td>
<td>✓ (CFI)</td>
<td>Argcount mismatch</td>
</tr>
<tr>
<td><strong>COOP ML-REC [13]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Chrome</td>
<td>✓ (CFI)</td>
<td>Argcount mismatch,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Void target where non-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void was expected</td>
</tr>
<tr>
<td><strong>Control Jujutsu [16]</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Apache</td>
<td>✓ (CFI)</td>
<td>Target function not AT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Void target where non-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void was expected</td>
</tr>
</tbody>
</table>
Stopping COOP Exploits in Practice

Exploit on 64-bit IE

Figure 6. Gadgets used in COOP’s 64-bit IE exploit
Stopping COOP Exploits in Practice

- 64-bit IE
- 64-bit Firefox
  - ML-G used for IE and Firefox attack prepares only one argument
  - most vfgadgets use at least two arguments
- Chrome
  - a callsite is non-void but one of the chained vfgadgets is of type void
Conclusion

- applies type-based invariants, inspired by source-level CFI techniques, at the binary level for the first time
- successfully mitigates all publicly available exploits that are not pure data-only attacks
Possible Future Work

- Advanced argument-passing techniques (already been tackled by source-level CFI solutions)
- Pure Data-only Attacks (like Control Flow Bending) is still not mitigated

Thank you very much for your attention!

Any questions/remarks/suggestions?