Counterfeit Object-oriented Programming

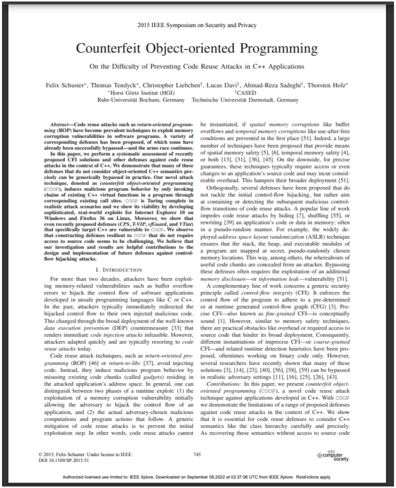
On the Difficulty of Preventing Code Reuse Attacks in C++ Applications

Raghuveer Shivakumar (Raghu)

University of Colorado, Boulder

2022-09-07

Counterfeit Object-oriented Programming



2015 IEEE Symposium on Security and Privacy

Counterfeit Object-oriented Programming

On the Difficulty of Preventing Code Reuse Attacks in C++ Applications

Felix Schuster*, Thomas Tendyck*, Christopher Liebchen†, Lucas Davi†, Ahmad-Reza Sadeghi†, Thorsten Holz*
*Horst Görtz Institut (HGI), Ruhr-Universität Bochum, Germany
†CASED, Technische Universität Darmstadt, Germany

***Abstract*—Code reuse attacks such as *return-oriented programming* (ROP) have become prevalent techniques to exploit memory corruption vulnerabilities in software programs. A variety of corresponding defenses has been proposed, of which some have already been successfully bypassed—and the arms race continues.**

In this paper, we perform a systematic assessment of recently proposed CFI solutions and other defenses against code reuse attacks in the context of C++. We demonstrate that many of these defenses that do not consider object-oriented C++ semantics precisely can be generically bypassed in practice. Our novel attack technique, denoted as *counterfeit object-oriented programming* (COOP), induces malicious program behavior by only invoking chains of existing C++ virtual functions in a program through corresponding existing call sites. COOP is Turing complete in realistic attack scenarios and we show its viability by developing sophisticated, real-world exploits for Internet Explorer 10 on Windows and Firefox 36 on Linux. Moreover, we show that even recently proposed defenses (*CPS*, *T-VIP*, *vfGuard*, and *VTint*) that specifically target C++ are vulnerable to COOP. We observe that constructing defenses resilient to COOP that do not require access to source code seems to be challenging. We believe that our investigation and results are helpful contributions to the design and implementation of future defenses against control-flow hijacking attacks.

I. Introduction

For more than two decades, attackers have been exploiting memory-related vulnerabilities such as buffer overflow errors to hijack the control flow of software applications developed in unsafe programming languages like C or C++. In the past, attackers typically immediately redirected the hijacked control flow to their own injected malicious code. This changed through the broad deployment of the well-known *data execution prevention* (DEP) countermeasure [33] that renders immediate *code injection attacks* infeasible. However, attackers adapted quickly and are typically resorting to *code reuse attacks* today.

Code reuse attack techniques, such as *return-oriented programming* (ROP) [46] or *return-to-libc* [37], avoid injecting code. Instead, they induce malicious program behavior by misusing existing code chunks (called *gadgets*) residing in the attacked application's address space. In general, one can distinguish between two phases of a runtime exploit: (1) the exploitation of a memory corruption vulnerability initially allowing the adversary to hijack the control flow of an application, and (2) the actual adversary-chosen malicious computations and program actions that follow. A generic mitigation of code reuse attacks is to prevent the initial exploitation step. In other words, code reuse attacks cannot be instantiated, if *spatial memory corruptions* like buffer overflows and *temporal memory corruptions* like use-after-free conditions are prevented in the first place [51]. Indeed, a large number of techniques have been proposed that provide means of spatial memory safety [5], [6], temporal memory safety [4], or both [13], [31], [36], [45]. On the downside, for precise guarantees, these techniques typically require access or even changes to an application's source code and may incur considerable overhead. This hampers their broader deployment [51].

Orthogonally, several defenses have been proposed that do not tackle the initial control-flow hijacking, but rather aim at containing or detecting the subsequent malicious control-flow transitions of code reuse attacks. A popular line of work impedes code reuse attacks by hiding [7], shuffling [55], or rewriting [39] an application's code or data in memory; often in a pseudo-random manner. For example, the widely deployed *address space layout randomization* (ASLR) technique ensures that the stack, the heap, and executable modules of a program are mapped at secret, pseudo-randomly chosen memory locations. This way, among others, the whereabouts of useful code chunks are concealed from an attacker. Bypassing these defenses often requires the exploitation of an additional *memory disclosure*—or *information leak*—vulnerability [51].

A complementary line of work concerns a generic security principle called *control-flow integrity* (CFI). It enforces the control flow of the program to adhere to a pre-determined or at runtime generated control-flow graph (CFG) [3]. Precise CFI—also known as *fine-grained* CFI—is conceptually sound [1]. However, similar to memory safety techniques, there are practical obstacles like overhead or required access to source code that hinder its broad deployment. Consequently, different instantiations of imprecise CFI—or *coarse-grained* CFI—and related runtime detection heuristics have been proposed, oftentimes working on binary code only. However, several researchers have recently shown that many of these solutions [3], [14], [23], [40], [56], [58], [59] can be bypassed in realistic adversary settings [11], [16], [25], [26], [43].

Contributions: In this paper, we present *counterfeit object-oriented programming* (COOP), a novel code reuse attack technique against applications developed in C++. With COOP we demonstrate the limitations of a range of proposed defenses against code reuse attacks in the context of C++. We show that it is essential for code reuse defenses to consider C++ semantics like the class hierarchy carefully and precisely. As recovering these semantics without access to source code

© 2015, Felix Schuster. Under license to IEEE.
DOI 10.1109/SP.2015.51

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 06,2022 at 02:37:06 UTC from IEEE Xplore. Restrictions apply.

Authors: Felix Schuster, Thomas Tendyck, Christopher Liebchen, Licas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz

Description: Using language semantics and counterfeit objects to bypass code-reuse defenses in C++ applications

Background - Attacks

- **Code reuse attacks** are a class of attacks that exploit the reuse of code in a program to execute malicious code
  - Example: Return-oriented Programming (ROP), Return-to-libc (RTL), etc.
  - ```
    char* vulnerable(char* input) {
      char buffer[10];
      strcpy(buffer, input);
      return buffer;
    }
    ```
    - Here the attacker can overwrite the return address to point to malicious code
- **Counterfeit object-oriented programming** (COOP) is a code reuse attack that exploits the reuse of object-oriented code in a program to execute malicious code. Especially, using the polymorphic feature of the language to execute malicious code.
    ```

Background - Attacks (cont.)

Most of these attacks are based on:

- indirect calls/jumps to non address-taken locations
- returns not in compliance with the call stack
- pivoting of the stack pointer (possibly temporarily)
- injection of new code pointers or manipulation of existing ones

Heavily dependent on control flow or data flow.

Background - Mitigations for Non-COOP Attacks

- **CFI**, **Stack Canaries**, **Shadow Stacks**, **ASLR**, **CPI**, etc are some of the mitigation techniques that are used to prevent code reuse attacks.

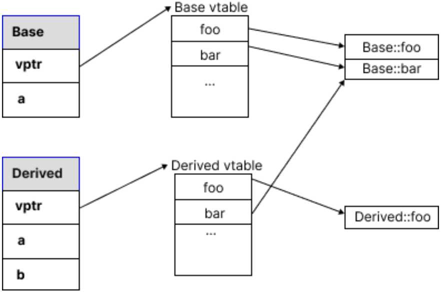
Not enough to prevent COOP attacks. We'll see why.

Hint: Language semantics play a vital role in COOP attacks.

Background - C++ Object Model

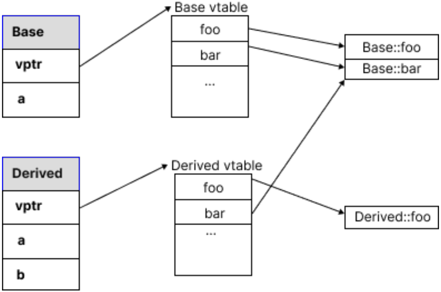
Let's understand with an example:

```
struct Base {
  int a;
  virtual void foo()
  { cout << "Base::foo"; }
  virtual void bar()
  { cout << "Base::bar"; }
};
struct Derived : public Base {
  int b;
  void foo()
  { cout << "Derived" << endl; }
  ~Derived()
  { cout << "Destroy Derived" << endl; }
};
```



Background (contd.) - C++ Object Model (contd.)

- **Address-taken functions**: Functions that have a constant pointer pointing to them
  - Virtual functions are address-taken functions
- Virtual Function Call
  - `mov rdx, qword ptr [rcx]` `call qword ptr [rdx+8]`
  - Compiler generally hardcode the index at a call site.
    - Here it is always the second entry in the vtable.



COOP Attack - Overview

Assumptions

- The attacker has control of some object with vptr.
- The attacker can infer the base address of a set of c++ modules whose binary layout is known to them.
  - In practice, it is expected that the attacker is aware of the presence of one or more of publicly available c++ libraries in the code base and can infer the base address of these libraries.

COOP Attack - Key Concepts

- **Counterfeit Object**: An object that is not part of the program but is created by the attacker to be used in the program.
- **Vfgadgets**: A set of virtual functions to assist in creating a COOP attack.

Vfgadget type	*Purpose*	*Code example*
ML-G	The main loop; iterate over container of pointers to counterfeit object and invoke a virtual function on each such object.	see Figure 1
ARITH-G	Perform arithmetic or logical operation.	see Figure 4
W-G	Write to chosen address.	see Figure 4
R-G	Read from chosen address.	no example given, similar to W-G
INV-G	Invoke C-style function pointer.	see Figure 8
W-COND-G	Conditionally write to chosen address. Used to implement conditional branching.	see Figure 6
ML-ARG-G	Execute vfgadgets in a loop and pass a field of the *initial object* to each as argument.	see Figure 6
W-SA-G	Write to address pointed to by first argument. Used to write to *scratch area*.	see Figure 6
MOVE-SP-G	Decrease/increase stack pointer.	no example given
LOAD-R64-G	Load argument register `rdx`, `r8`, or `r9` with value (x64 only).	see Figure 4

TABLE I: Overview of COOP vfgadget types that operate on object fields or arguments; general purpose types are atop; auxiliary types are below the double line.

COOP Attack - MLG-Based Attack

- Use counterfeit objects to point to arbitrary entry in the vtable using the Main loop.
  - Example counterfeit object: `students` in the figure below.

```
class Student {
public:
    virtual void incCourseCount() = 0;
    virtual void decCourseCount() = 0;
};

class Course {
private:
    Student **students;
    size_t nStudents;
public:
    /* ... */
    virtual ~Course() {
        for (size_t i = 0; i < nStudents; i++)
            students[i]->decCourseCount();          ML-G
        delete students;
    }
};
```

Fig. 1: Example for ML-G: the virtual destructor of the class `Course` invokes a virtual function on each object pointer in the array `students`.

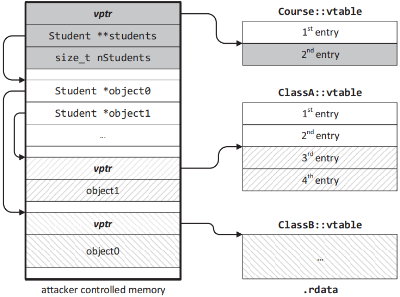


Fig. 2: Basic layout of attacker controlled memory (left) in a COOP attack using the example ML-G `Course::~Course`. The initial object (dark gray, top left) contains two fields from the class `Course`. Arrows indicate a *points-to* relation.

Not so useful in practice, as most of the real-world attacks require that the attacker be able to not only control object fields, but also arguments to the vfgadgets.

COOP Attack - MLG with Overlapping Objects

- Use overlapping objects to control both object fields and arguments to the vfgadgets.

```
class Exam {
private:
    size_t scoreA, scoreB, scoreC;
public:
    /* ... */
    char *topic;
    size_t score;
    virtual void updateAbsoluteScore() {
        score = scoreA + scoreB + scoreC;
    }                                              ARITH-G

    virtual float getWeightedScore() {
        return (float)(scoreA*5+scoreB*3+scoreC*2) / 10;
    }
                                                LOAD-R64-G
};

struct SimpleString {
    char* buffer;
    size_t len;
    /* ... */
    virtual void set(char* s) {
        strncpy(buffer, s, len);                       W-G
    }
};
```

Fig. 4: Examples for ARITH-G, LOAD-R64-G, and W-G; for simplification, the native integer type `size_t` is used.

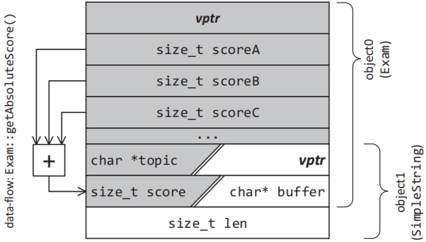


Fig. 5: Overlapping counterfeit objects of types `Exam` and `SimpleString`

COOP Attack - MLG with Overlapping Objects (contd.)

- Passing arguments to the vfgadets by hijacking the registers.
- **Example Calling Convention (Win64):**
  - this-ptr = `rdi`, arg1 = `rdx`, arg2 = `r8`, arg3 = `r9`, arg4 and on = `stack`

Example LOAD-R64 Gadget:

```
virtual float getWeightedScore() {
    return (float)(scoreA*5+scoreB*3+scoreC*2) / 10;
}
```

LOAD-R64-G

In condensed from, this LOAD-R64-G provides the following useful semantics to the attacker:

$$\texttt{rdx} \leftarrow 3 \cdot [\textit{this} + 10h]$$
$$\texttt{r8} \leftarrow [\textit{this} + 18h]$$
$$\texttt{r9} \leftarrow 3 \cdot [\textit{this} + 18h] + 2 \cdot [\textit{this} + 10h]$$

Generated Assembly for the LOAD-R64 Gadget:

As an example for a LOAD-R64-G, consider `Exam::getWeightedScore()` from Figure 4; MSVC compiles this function to the following assembly code:

```
mov rax, qword ptr [rcx+10h]
mov r8, qword ptr [rcx+18h]
xorps   xmm0, xmm0
lea rdx, [rax+rax*2]
mov rax, qword ptr [rcx+8]
lea rcx, [rax+rax*4]
lea r9, [rdx+r8*2]
add r9, rcx
cvtsi2ss    xmm0, r9
addss   xmm0, dword ptr [__real0]
divss   xmm0, dword ptr [__real1]
ret
```

COOP Attack - Main Motivation

- Exploit Windows API (WinAPI) functions such as WinExec() or VirutalProtect()
- Examples:
  - use a vfgadget that legitimately calls WINAPI function of interest.
    - Mostly not feasible.
  - invoke the WINAPI function like a virtual function from the COOP main loop.
    - Make vptr point to IAT/EAT instead of vtable.

Framework for COOP Attacks

Steps:

- Identify the vfgadet
- Implementation fo attack semantics using the identified vfgadets
- Arrangement of possibly overlapping counterfeit objects in a buffer.

Doing this manually is tedious and error-prone. Authors have implemented a python-based framework for performing (i) and (ii) automatically.

Identification

- vfgadget finder
  - Binary x84-64 c++ modules are disassembled.
  - Each virtual function is considered a potential gadget.
  - Each address address taken array of function pointers is considered a potential vtable.
  - As a heuristic, pick functions with limited number of basic blocks (usually 1 - 3).
  - Summarize the semantics of each basic block in SSA form.
    - Eg. *"left side of assignment must dereference any argument register; right side must dereference the this-ptr"*

Aligning overlapping objects

- Define counterfeit objects and labels
- Assign the label to the counterfeit objects
- Make sure that the label(id) offset of different objects are aligned using Z-3 solver.
  - $$offsetobj1 + labeloffsetobj1 = offsetobj2 + labeloffsetobj2$$

Real-World Example

Internet Explorer Vulnerability - CVE-2013-2551

- Read pointer to kernell32.dll from IAT
- Calculate pointer to WinExec() in kernel32.dll
- Read tick count from KUSER_SHARED_DATA data str
- if tick count is odd, launch calc, else launch mspaint.

Symbol name of vfgadget (mshtml.dll Win. 7 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
CExtendedTagNamespace::Passivate	1, 9b	ML-G	array-based main loop
CCircularPositionFormatFieldIterator::Next	2, 5, 7, 9a, 10b	LOAD-R64-G	load rdx from dereferenced field
XHDC::SetHighQualityScalingAllowed	3	ARITH-G	store rdx&1
CWigglyShape::OffsetShape	4	LOAD-R64-G	load r9 from field
CStyleSheetArrayVarEnumerator::MoveNextInternal	6	LOAD-R64-G	load r8 from field
CDataCache<class CBoxShadow>::InitData	8	W-COND-G	write r8 to [rdx] if r9 is not zero
CRectShape::OffsetShape	10a, 11b	ARITH-G	add [rdx] to field
Ptls6::CLsBlockObject::Display	11a, 12b	INV-G	invoke field as function pointer

TABLE A.I: Vfgadgets in mshtml.dll *10.0.9200.16521* used in exemplary Internet Explorer 10 64-bit exploit (§V-A); execution splits into paths *a* and *b* after index 8.

Symbol name of vfgadget (mshtml.dll Win. 7 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
CExtendedTagNamespace::Passivate	1	ML-G	array-based main loop
CMarkupPageLayout::IsTopLayoutDirty	2, 4	LOAD-R64-G	load edx from field
HtmlLayout::GridBoxTrackCollection::GetRangeTrackNumber	3	ARITH-G	$r8 = 2 \cdot rdx$
CAnimatedCacheEntryTyped<float>::UpdateValue	4	INV-G	invoke field from argument as function pointer

TABLE A.II: Vfgadgets in mshtml.dll *10.0.9200.16521* used in exemplary Internet Explorer 10 64-bit exploit that only uses vptrs pointing to the beginning of existing vtables (§V-A1)

Symbol name of vfgadget	*# in attack code*	*Vfgadget type*	*Function*
jscript9!ThreadContext:: ResolveExternalWeakReferencedObjects	1	ML-ARG-G	linked list-based main loop
CDataTransfer::Proxy	2	W-SA-G	write deref. field to scratch area
CDCompSwapChainLayer::SetDesiredSize	3	R-G	load field from scratch area
CDCompSurfaceTargetSurface::GetOrigin	4	ARITH-G and W-SA-G	write summation of two fields to scratch area
CDCompLayerManager:: SetAnimationCurveToken	5	R-G	load field from scratch area
HtmlLayout::SvgBoxBuilder:: PrepareBoxForDisplay	*loop_entry*: 6, 11	W-G	rewrite *argument field*
CDXTargetSurface::OnEndDraw	7, 8	MOVE-SP-G	move stack pointer up
ieframe!Microsoft::WRL:: Callback::ComObject::Invoke	9	INV-G	invoke function pointer with 2 arguments
CMarkupPageLayout::AddLayoutTaskOwnerRef	10	ARITH-G	increment field
Ptls6::CLsDnodeNonTextObject:: SetDurFmtCore	12	W-COND-G	conditionally write argument to field; rewrites linked list; resumes at *loop_entry* or *loop_exit*
CDispRecalcContext:: OnBeforeDestroyInitialIntersectionEntry	*loop_exit*	NOP	nop; loops to self

TABLE A.III: Vfgadgets used in exemplary Internet Explorer 10 32-bit exploit (§V-B); vfgadgets taken from mshtml.dll (if not marked differently), jscript9.dll, or ieframe.dll version *10.0.9200.16521*.

Symbol name of vfgadget (libxul.so Linux 64-bit)	*# in attack code*	*Vfgadget type*	*Function*
nsMultiplexInputStream::Close	1	ML-G	array-based main loop
mozilla::a11y::xpcAccessibleGeneric::~xpcAccessibleGeneric and js::jit::MVariadicInstruction::getOperand	2, 4	LOAD-R64-G	load rsi from memory
nsDisplayItemGenericGeometry::MoveBy	3	ARITH-G	add [rsi] to field
ProfileSaveEvent::AddSubProfile	5	INV-G	invoke field as function pointer

TABLE A.IV: Vfgadgets used in exemplary Firefox 36.0a1 64-bit exploit (§V-C)

Other experiments include one on Firefox, and exploiting other similary vulnerabilities in Internet Explorer.

Defences against COOP Attacks - Generic Defences

- *Binary Rewriting:* make sure that certain WINAPI functions may only be invoked using multiple indirect calls.
- *Fine Grained Code Randomization:* LOAD-R64-G can be broken using this as the location of the argument is also randomized.

Defences against COOP Attacks - COOP-Specific Defences

- *Verificaiton of vptrs:* All vcall sites are known beforehand and the vptr access can be verified with sanity checks. (Reliability?)
- *Monitor Data Flow:* Monitor the objects in MLG. Extremely difficulty as the tool needs to be able to track the objects throughout their lifetime. (Reliability?)
- *Fine grained randomization of C++ data structures:* The layout of each counterfeit object needs to be byte compitable with the semantics of its vfgadets. Add random padding to the object fields make this harder for the attacker.

Real-world Defence Analysis

Category	Scheme	Realization	Effective against COOP ?
Generic CFI	Original CFI + shadow call stack [3]	Binary + debug symbols	✗
	CCFIR [58]	Binary	✗
	O-CFI [54]	Binary	✗
	SW-HW Co-Design [15]	Source code + specialized hardware	✗
	Windows 10 Tech. Preview CFG	Source code	✗
	LLVM IFCC [52]	Source code	?
C++-aware CFI	**—various—** [5], [29], [52]	Source code	✓✓✓
	T-VIP [24]	Binary	✗
	VTint [57]	Binary	✗
	vfGuard [41]	Binary	?
Heuristics-based detection	**—various—** [14], [40], [56]	CPU debugging/performance monitoring features	✗✗✗
	HDROP [60]	CPU performance monitoring counters	✗
	Microsoft EMET 5 [34]	WinAPI function hooking	✗
Code hiding, shuffling, or rewriting	STIR [55]	Binary	✗
	G-Free [38]	Source code	✗
	XnR [7]	Binary / source code	?
Memory safety	**—various—** [4]–[6], [13], [36], [45]	Mostly source code	(✓✓✓) - see §VII-E
	CPI/CPS [31]	Source code	✓/✗

TABLE II: Overview of the effectiveness of a selection of code reuse defenses and memory safety techniques (below double line) against COOP; ✓ indicates effective protection and ✗ indicates vulnerability; ? indicates at least partial protection.

Summary

- COOP attacks are a new class of attacks that exploit the C++ object model.
- They are particularly effective because of their ability to levergage language semantics to bypass existing defences.
- Defences against existing Code Reuse attacks do not fare well against COOP attacks.
  - You need C++-semantic-aware defences that can detect and prevent COOP attacks.

Questions