It’s a TRaP: Table Randomization and Protection against Function-Reuse Attacks

Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, Michael Franz

Presented by: Jacob Venne
Outline

• Goals
• Technical background
  ▫ Virtual function calls
  ▫ Dynamic Linking
• Strengthening COOP Attack
• Mitigation Goals
• Readactor++
• Countering Memory Disclosure
• Vtable Randomization
• Procedure Linkage Table Randomization
• Countering Guessing Attacks
Goals

- **Problem:** Counterfeit Object Oriented Programming (COOP) attacks bypasses all published control-flow integrity systems
- **Goals:** solve problems in existing randomization approaches by preventing COOP or Return-into-libc (RILC) attacks
  - Permute tables containing code pointers while preserving semantics
  - Inserting booby traps to mitigate probing attacks
  - Prevent memory disclosure by transforming tables to use execute-only page permissions
Virtual Function Calls

- Used to support polymorphism
  - normal functions are \textit{statically-bounded} $\rightarrow$ function is known at build time.
  - virtual functions are \textit{dynamically-bounded} $\rightarrow$ function is known at runtime and is dependent on the type of object.
- When virtual functions are called, the correct function is chosen from a readable/writable table called a vtable.
- Adversaries abuse indirect calls to virtual functions by corrupting objects to point to a vtable of their choice, allowing them to control the call destination.
Dynamic Linking

- Allows programmers to group functions and classes with related functionality into a single module
- Each module has a set of symbols exposed to other modules through a symbol table
- Symbol addresses are usually kept in writable/readable memory
- These tables are vulnerable to information leakage attacks
Dynamic Linking Vulnerability

- The Global Offset Table (GOT) stores addresses of global and static elements including the addresses of all of all imported functions for use in Procedure Linkage Table (PLT).
- Attacker can collect function addresses from GOT.
- The PLT contains a set of trampolines that each correspond to a code pointer in the GOT.
- Using relocation, symbol and string tables, an attacker can infer the layout of both GOT and PLT tables.
  - trampolines: direct jumps with the same target as the code pointer it replaces.
Code Reuse Attack

- Data Execution Prevention (DEP) prevents code injection so attackers use code-reuse also known as return-into-libc (RILC) attacks.
- COOP code-reuse attacks hijack the control flow and redirect it to a sequence of virtual function calls on a set of C++ objects.
- Attacker must know exact layout of the relevant objects and vtables.
- COOP relies on a main loop gadget (ML-G) defined as “A virtual function that iterates over a container of pointers to C++ objects and invokes a virtual function on each of these objects”.
- ML-G is the first virtual function executed in this attack and it’s role is to start the other virtual functions called vfgadgets.
Recursive COOP to Emulate ML-G

- Recursive gadget – REC-G
- ~Z() called on adversary controlled object allows attacker to control pointers objA and objB.
- Allows adversary to repeatedly invoke the REC-G on itself to build an arbitrary number of vfgadgets.
- Functions don’t take arguments which allow attacker to use these registers to pass arguments between vfgadgets.
- Any virtual function that invokes virtual functions on 2 or more object pointers can be used as REC-G
Unrolled COOP to Emulate ML-G

- Unrolled gadget – UNR-G
- Given a function with many virtual function calls.
- C::func() can be misused on counterfeit objects to create 4 vfgadgets.
- The execution of three vfgadgets is enough to execute arbitrary code.

```cpp
void C::func() {
    delete obj0; delete obj1; delete obj2; delete obj3;
}
```
How to Mitigate Function-Reuse

- Defensive requirements:
  - Writeable or Executable memory but not both - Prevents attack from injecting/modify executable code
  - Execute-Only Memory – normal read/write accesses trigger access violation
  - Brute-force mitigation - software does not automatically restart after hitting a booby trap.

- Readactor++
Readactor++

Attack Requirements:
• COOP payload requires knowing the exact representation of objects and vtables.
• LIBC attacks require layout knowledge of PLT tables

Readactor Goals:
• Readactor++ aims to permute and hide these tables even if the attacker knows the arbitrary readable memory.
• Done by compiler
Readactor++ Steps

1. Ensure that code can be mapped in execute-only memory and perform code-pointer hiding (prevents inferring code layout from pointers and return addresses).
   - Split virtual tables into a read-only part (rvtable) and an execute-only part (xvtable).
   - Randomize these tables at load time.

2. Insert booby trap entries to the xvtable
   - Booby traps are code snippets that lie dormant during normal program execution and terminate the program and alert the host system if they are ever invoked.

3. Collects metadata called Translation and Protection (TRaP)
   - Embed TRaP information in output binaries

4. Link randomization engine RandoLib into the output binary.
Readactor++ Model

- Protected App: main executable and program library contain TRaP info read by RandoLib to randomize vtables and PLT at load time.
- Protected Process: Global variable, program stacks, and heap stored in read/write memory.
- vtables split into read only (rvtable) and write only (wvtable) parts.
  - rvtables store pointer to corresponding vtable instead of direct code pointer
  - xvtable contains direct jumps instead of code pointers to allow execute-only
Protection Against Memory Disclosure

- Readactor++ offers protection against indirect memory disclosure
  - Indirect memory disclosure: adversary reads code pointers stored in program vtables, stacks, and heaps to infer code layout
- Code-pointer hiding replaces all code pointers in memory with pointers to trampolines.
- Trampolines are direct jumps to the target of original code pointer
- This prevents adversaries from dereferencing code pointers and finding code-layout
Vtable Randomization

**Goal:** the layout of vtables
- The vtable is split into two segments, xvtable and rvtable
- A pointer xpointer is added to the rvtable that points to the xvtable
- Once the vtable has been split, we can randomize the order of functions in the xvtable
- Virtual calls are accessed using a static offset that correlates to the new randomized vtable
A call to a dynamically linked function goes through the PLT of the calling module.

PLT entries load code pointer from GOT into a register and transfers control to the target function through an indirect jump.

Readactor++ randomizes the order of PLT, inserts booby traps, and store the result in execute-only memory, allowing us to discard the GOT.

PLT is rewritten to jump to the target function directly instead of reading the address from GOT and indirectly jumping.
Countering Guessing Attacks

- Brute-force attacks are powerful against services that automatically restart without randomization.
- Readactor++ inserts booby traps to detect probing.
- If adversary runs into a booby trap the program terminates immediately.
- The program re-randomizes when it starts up again, forcing the adversary to start over.
- The adversary would have to guess correct at least 2 times to create a gadget (.4%).
Sources