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

WRITTEN BY: CRANE ET AL.

PRESENTED BY: BRET FINLEY

Memory Corruption Attacks

Known as stack-smashing

Exploit data stored in RAM

- Execute data as machine instructions
- Buffer-Overflow corrupts and overwrites legitimate data
- Overwriting return address hijacks the Program Counter (PC)

Prevalent in languages with manual memory allocation/deallocation

- Most notably C and C++

Code Injection Attack

Supplants legitimate code with malicious code

Malicious code introduced into system from outside source

Overflow a buffer with malicious, carefully crafted code

Hijacked program counter returns into injected code

Code Reuse Attack

Uses pre-existing system or library code, rather than injected code

Once again, a buffer is overflowed to hijack program counter

Return-to-Libc - famous Code Reuse Attack

- Libc contains many system utility functions

ROP Attack

“Return Oriented Programming”

A variety of Code Reuse Attack

Once the PC is hijacked, the attacker employs a sequence of machine instructions called “gadgets”

Each gadget terminates with return instruction (ret)

- Transfers control to the next gadget

Gadgets chained together to execute a sequence of subroutines

JIT-ROP

"Just-In-Time Return-Oriented-Programming"

Developed to overcome address randomization

Abuse memory disclosure in order to map memory

Trick machine into disclosing the new randomized layout

Construct and compile malicious application on the fly

COOP Attack

"Counterfeit Object-Oriented Programming"

Exploits a vulnerability in C++ inheritance

External library calls are stored and looked up at runtime

- Enables method selection at runtime

Semantically similar to Return-to-Libc, but exploits virtual (inherited) functions

The Fall of Code Injection

Data Execution Prevention (DEP) resulted in Code Injection disuse

- “noexecstack” in linux
- Prevents data being interpreted as machine instructions
- Defeats Code Injection

Address randomization further complicates Code Injection

Increased the popularity of Code Reuse Attacks

Defending Against Reuse Attacks

Address Randomization

- Can’t assume address layout
- Base address random between executions
- Not a silver bullet
  - Just-In-Time Compilation learns address layout dynamically

Readactor Security Framework

- Defends against JIT-ROP attacks
- Hides code pointers and late-binding code from attackers
- Cannot stop COOP attacks

Motivation

Want to bolster randomization defenses for Code Reuse Attacks

- RILC
- COOP

Three contributions

- Obfuscate object tables containing code pointers
- Lay traps to discourage and diminish probing attacks
- Modify virtual tables from read-access to execute-access - preventing layout disclosure

Inheritance in C++

C++ supports multiple inheritance

Inherited methods explicitly defined with the *virtual* keyword

Non-virtual methods may not be overridden by subclasses

Virtual methods have bodies, pure virtual (abstract) methods do not

Non-virtual methods are compiled statically and called directly

Virtual method calls depend on the calling context

Virtual Methods

Virtual method calls rely on a pointer to a Vtable (virtual table)

The VTable contains overridden versions of a method

Correct entry selected when the function is called

```
class A {
    public:
        A(){}
        virtual void foo() {
            cout << "Override me!" << endl;
        }

        void bar() {
            cout << "Can't override me" << endl;
        }
};

class B : public A {
    public:
        B(){}
        void foo() {
            cout << "I'm class B!" << endl;
        }
};

class C : public A {
    public:
        C(){}
        void foo() {
            cout << "I'm class C!" << endl;
        }
};
```

```
int main(int argc, char *argv[]) {
    A *a;
    B b;
    C c;

    string cmp = "B";

    if(argv[1] == cmp)
    {
        a = &b;
    }

    else
    {
        a = &c;
    }

    a->foo();
    a->bar();

    return 0;
}
```

Function Linking at Runtime

Dynamic linking provides late binding between symbols and their routines

Varied program behavior from a single function call

Symbols-to-function addresses are kept in tables, resolved (and even modified) at runtime

Exploiting VTables

Virtual tables contain function pointers stored in read-only memory

However, The pointer to the table itself is stored in writable memory

COOP attack replaces the legitimate VTable with a malicious VTable

To execute the attack, the attacker must

- Craft their own virtual table
- Overwrite existing VTable pointer to point to their virtual table

Exploiting VTables Cont.

Once done, following table lookups will point to the malicious table

Essentially, the attacker interposes their own virtual table

- Virtual method lookups will be directed here

With the VTable corrupted, the attacker may inject their own methods in place of legitimate methods

ML-G

“Main Loop Gadget”

Analogous to gadgets from ROP

“The first virtual function that is executed in a COOP attack and its role is to dispatch the other virtual functions, called *vfgadgets* that make up the COOP attack”

COOP cannot be nullified by removing potential ML-Gs

- Recursive COOP
- Unrolled COOP

Recursive COOP

Any virtual method which invokes further virtual methods is a candidate for attack

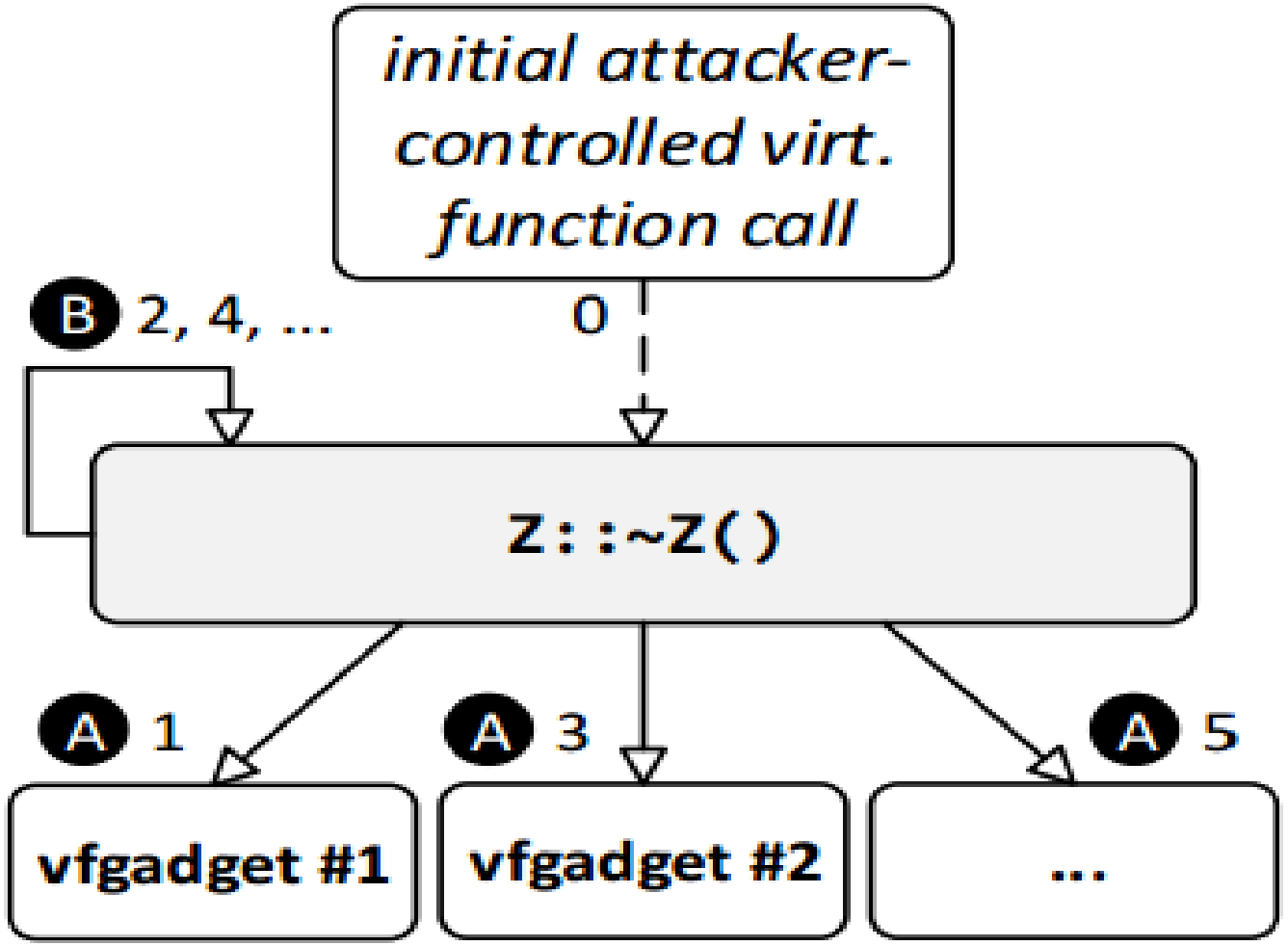
Destructors which call further destructors are a popular target

```
class X {
public:
  virtual ~X(); };

class Y {
public:
  virtual void unref(); };

class Z {
public:
  X *objA;
  Y *objB;
  virtual ~Z() {
     delete objA;      A
     objB->unref();    B
  } };
```

REC-G code example



control flow in REC-G-based COOP

Unrolled COOP

Possible to devise an attack without recursion or iteration

Two or more virtual calls provide an opportunity for an ML-G

Note that the destructors called here need to be virtual

```
void C::func() {
    delete obj0; delete obj1; delete obj2; delete obj3;
}
```

Security Requirements

Writable OR Executable, not both

- Table-splitting and randomization

Execute Only Instructions

- CPU may fetch and execute instructions, but reading and writing instructions is not allowed
- Store functions in execute-only memory

JIT Protection

- Protect against real-time memory disclosure
- Trampolines

Defense Against Brute Force and Probing

- Brute-Force attacks could incrementally reveal the randomized address space
- Booby-traps

Countering Attacks

“Show that probabilistic defenses can thwart function reuse attacks”

Their solution: Readactor++

A security framework aimed at combatting ROP attacks

Extended Readactor so it could defend against COOP attacks

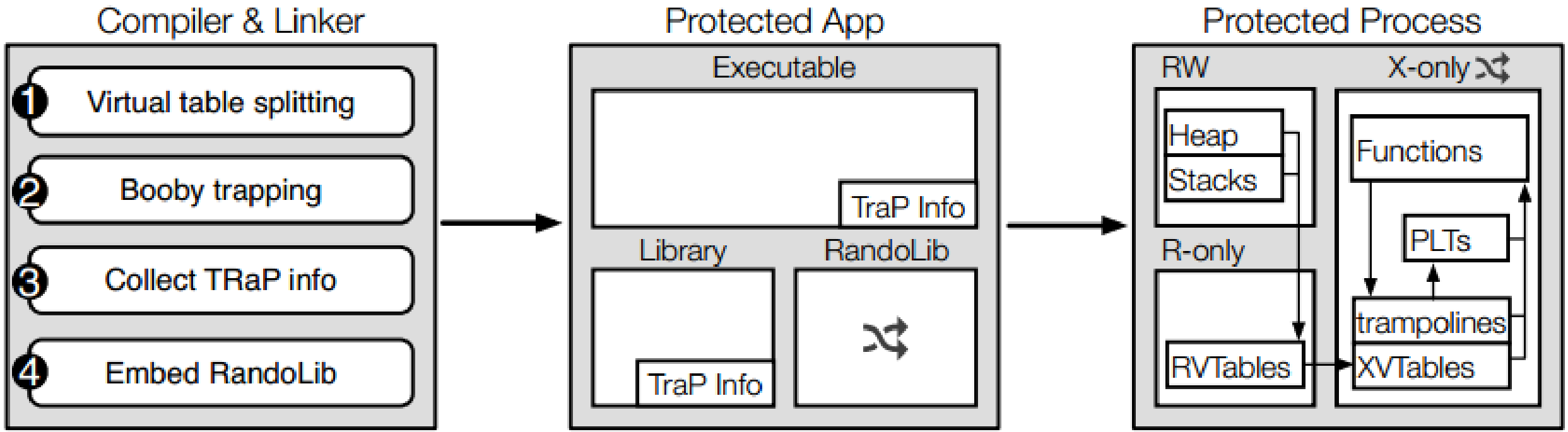
Countering Attacks Cont.

Readactor++ protects compiled binaries in four ways

- Virtual Table splitting
- Booby Trapping Execute-Only table entries
- Collect Translation and Protection metadata
- Randomize Virtual Table layout to make injection more difficult

Readactor++ took the form of a modified C++ compiler

Overview of Readactor++



Memory Disclosure

Attackers use tricks to force system to disclose memory layout

- Read code segment of the program stack
- Read code pointers from VTables, stack, or heap

Readactor++ hides all code pointers and replaces them with “trampolines”

Memory Disclosure Cont.

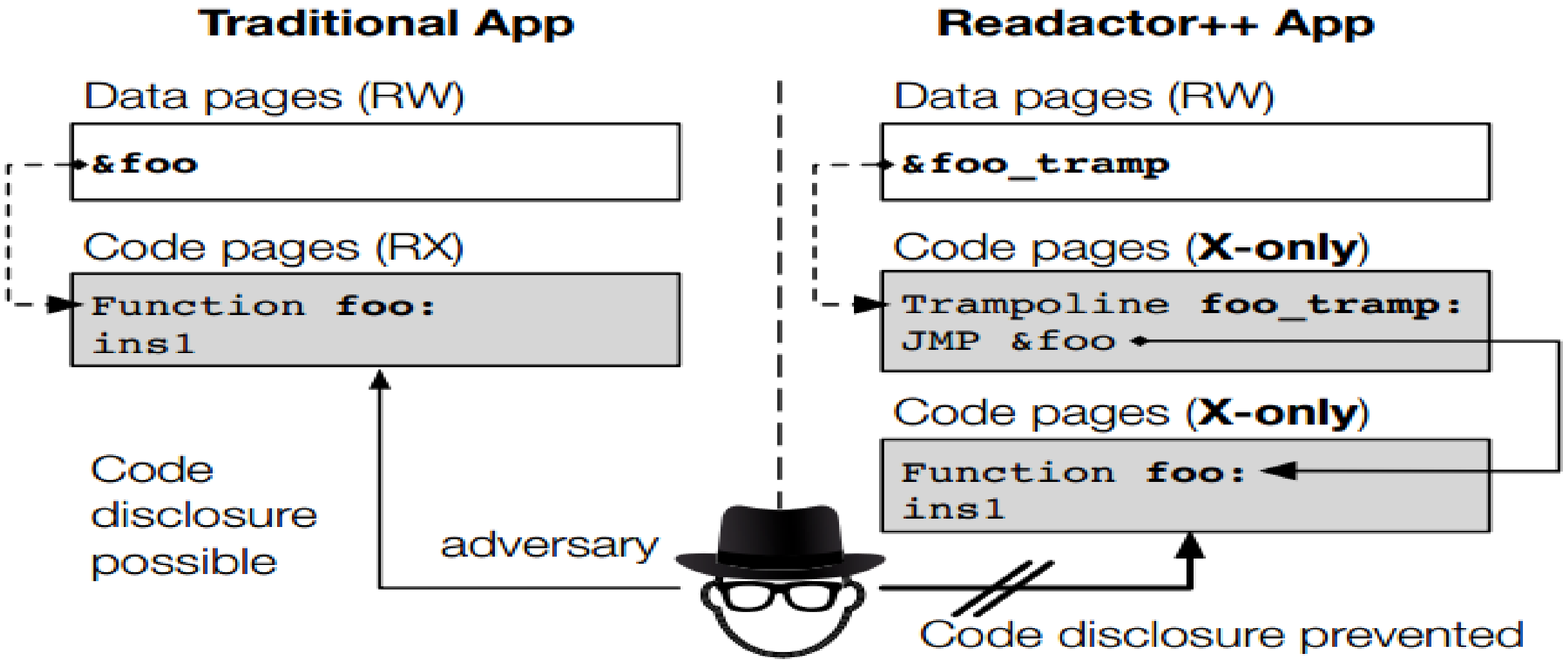
Trampoline - a jump instruction to the function pointer

Trampolines provide extra layer of indirection

Hides the code segment

Trampolines cannot be dereferenced (unlike pointers), attackers cannot force them to disclose memory layout

Trampolines



Virtual Table Randomization

Every C++ object contains metadata for pointing to its virtual table

- “vptr”

Randomizing object’s metadata layout is insufficient

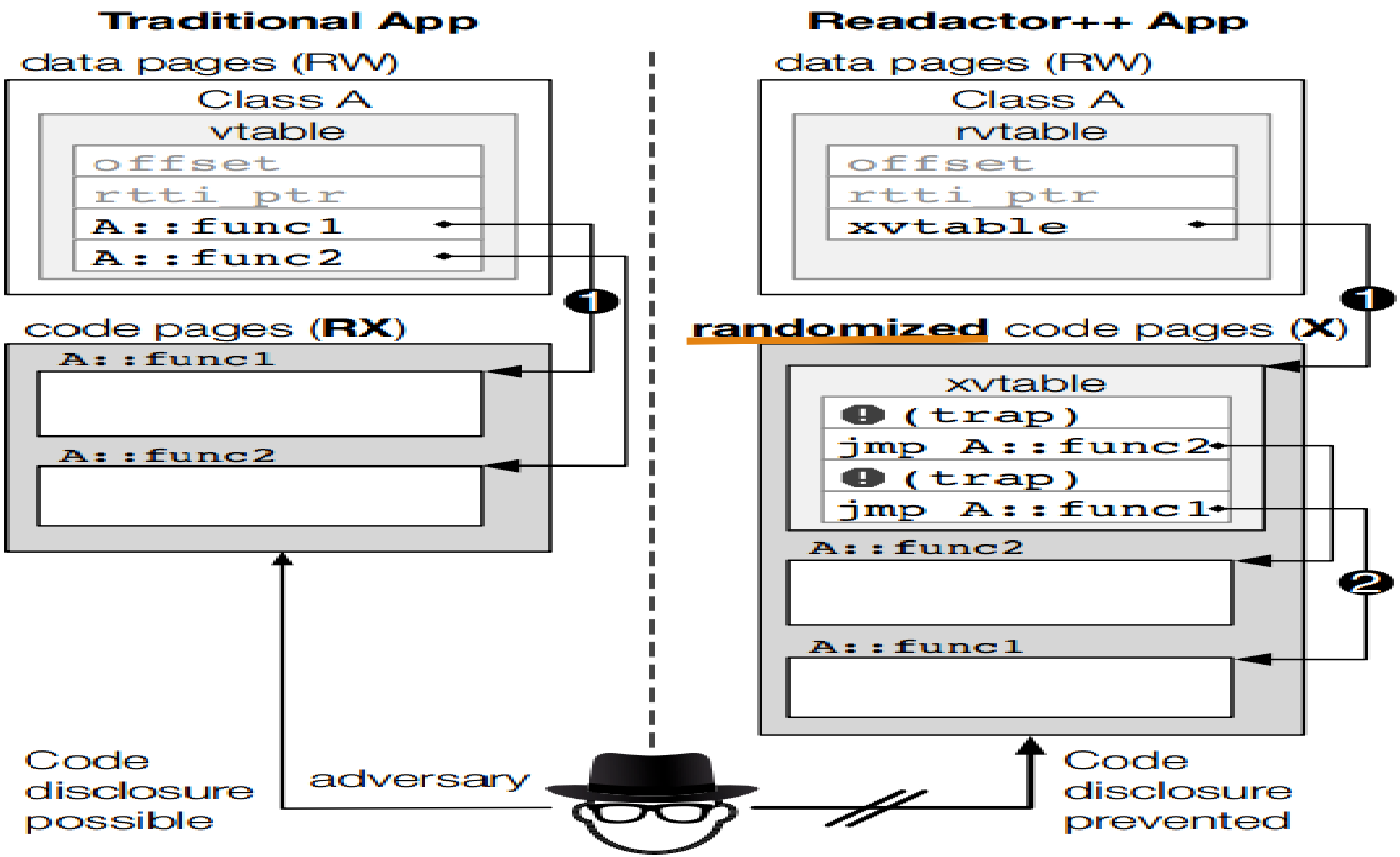
- Can be attacked like address randomization

Virtual Table Randomization Cont.

Virtual tables are better candidates for randomization

Once layout is randomized, need to prevent code pointers from being read

Solution: split the virtual tables into read-only and execute-only


Traditional App
data pages (RW)
Class A
vtable
offset
rtti_ptr
A::func1
A::func2
code pages (RX)
A::func1
A::func2
Readactor++ App
data pages (RW)
Class A
rvtable
offset
rtti_ptr
xvtable
randomized code pages (X)
xvtable
! (trap)
jmp A::func2
! (trap)
jmp A::func1
A::func2
A::func1
1
1
2
Code disclosure possible
adversary
Code disclosure prevented

Guessing Attacks

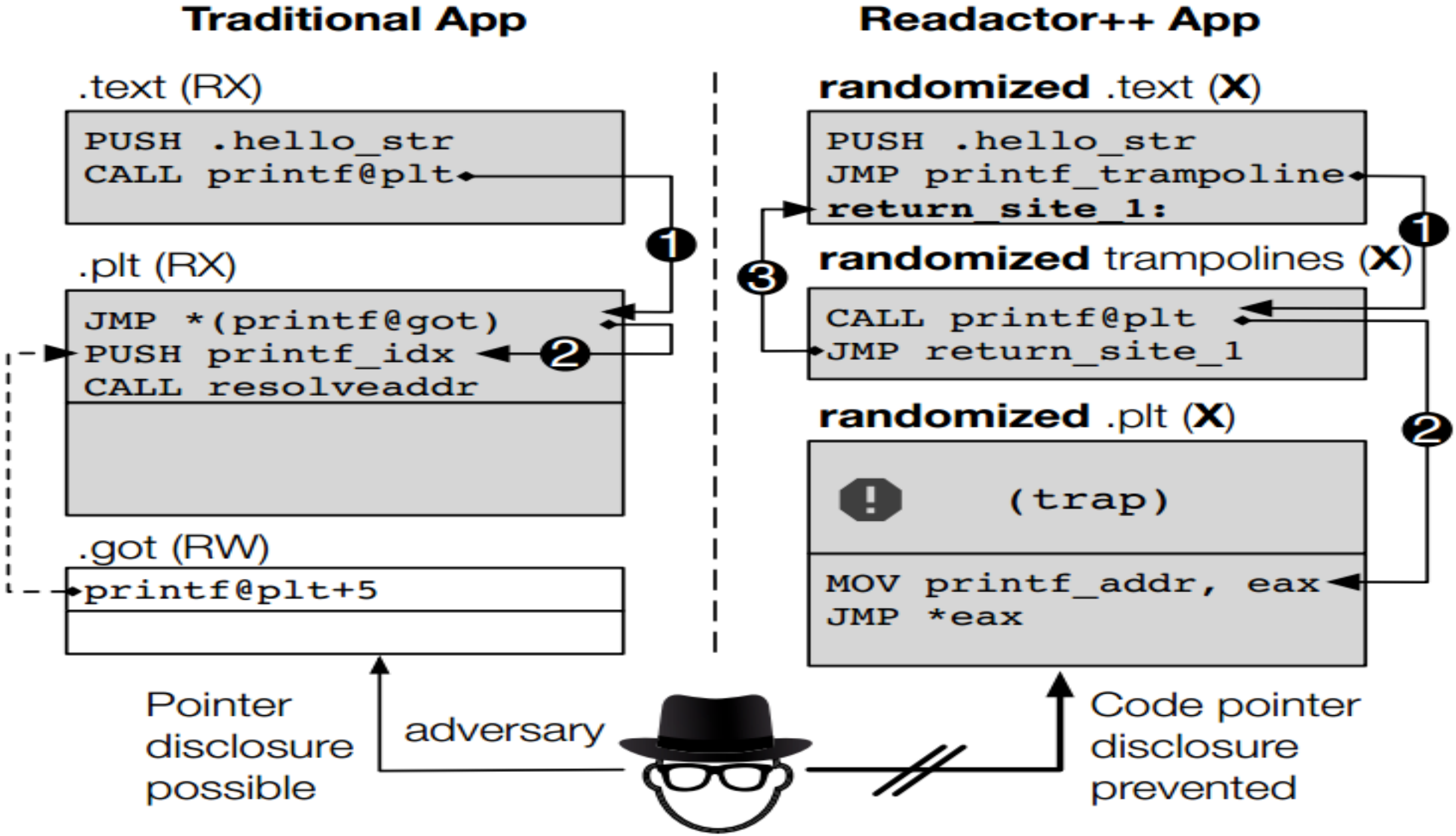
Brute force attacks exploit virtual tables

Attacker probes randomized VTable until familiar with layout

Booby-traps in execute-only VTable discourage probing

Triggering a booby trap

- Program will immediately terminate
- VTable layout will freshly randomize
- Application reports activity to system administrator


Traditional App
Readactor++ App
.text (RX)
PUSH .hello_str
CALL printf@plt
.plt (RX)
JMP *(printf@got)
PUSH printf_idx
CALL resolveaddr
.got (RW)
printf@plt+5
randomized .text (X)
PUSH .hello_str
JMP printf_trampoline
return_site_1:
randomized trampolines (X)
CALL printf@plt
JMP return_site_1
randomized .plt (X)
(trap)
MOV printf_addr, eax
JMP *eax
1
2
3
Pointer disclosure possible
adversary
Code pointer disclosure prevented

Procedure Linkage Table Randomization

Map for dynamically linked library functions

Exploiting this structure is vital to RILC attack's success

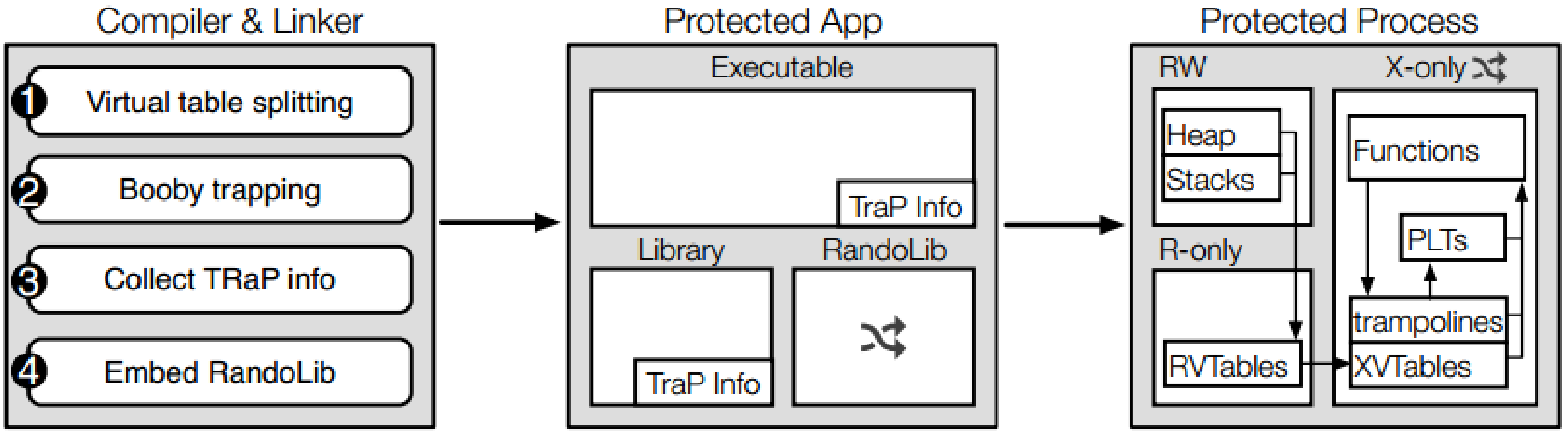
Entries organized in series, with base addresses randomized

Readactor++ randomizes entries, sets booby-traps to protect against exploitation

Trampolines ensure that function locations cannot be dereferenced

TRaP data is collected to update call instructions with the new, randomized function address

Overview of Readactor++



TRaP Metadata

“Table Randomization and Protection”

TRaP info denotes newly randomized locations of dynamically loaded functions and VTables

TRaP is specialized metadata that is compiled directly into the binary

Dynamic function and virtual calls must be updated with the new address

RandoLib

Randomization engine for assigning new function addresses

1. Alter table layouts and locations
2. Update call sites and references to change functions using the TRaP data

Security Performance

Trampoline addresses are still readable

- Possible security extension

Brute-Force Attacks

- Probe stack until layout is known
- Booby-traps stopped almost all brute-force attacks
- Success rate was 0.0003%

Practical Attacks – Chromium Web Browser

- JavaScript was used to create read/write locations for both COOP and RILC attacks
- Both attacks were successful on vanilla Chromium and failed on Chromium compiled with Readactor++

Runtime Efficiency Concerns

Chromium web browser was built using Readactor++

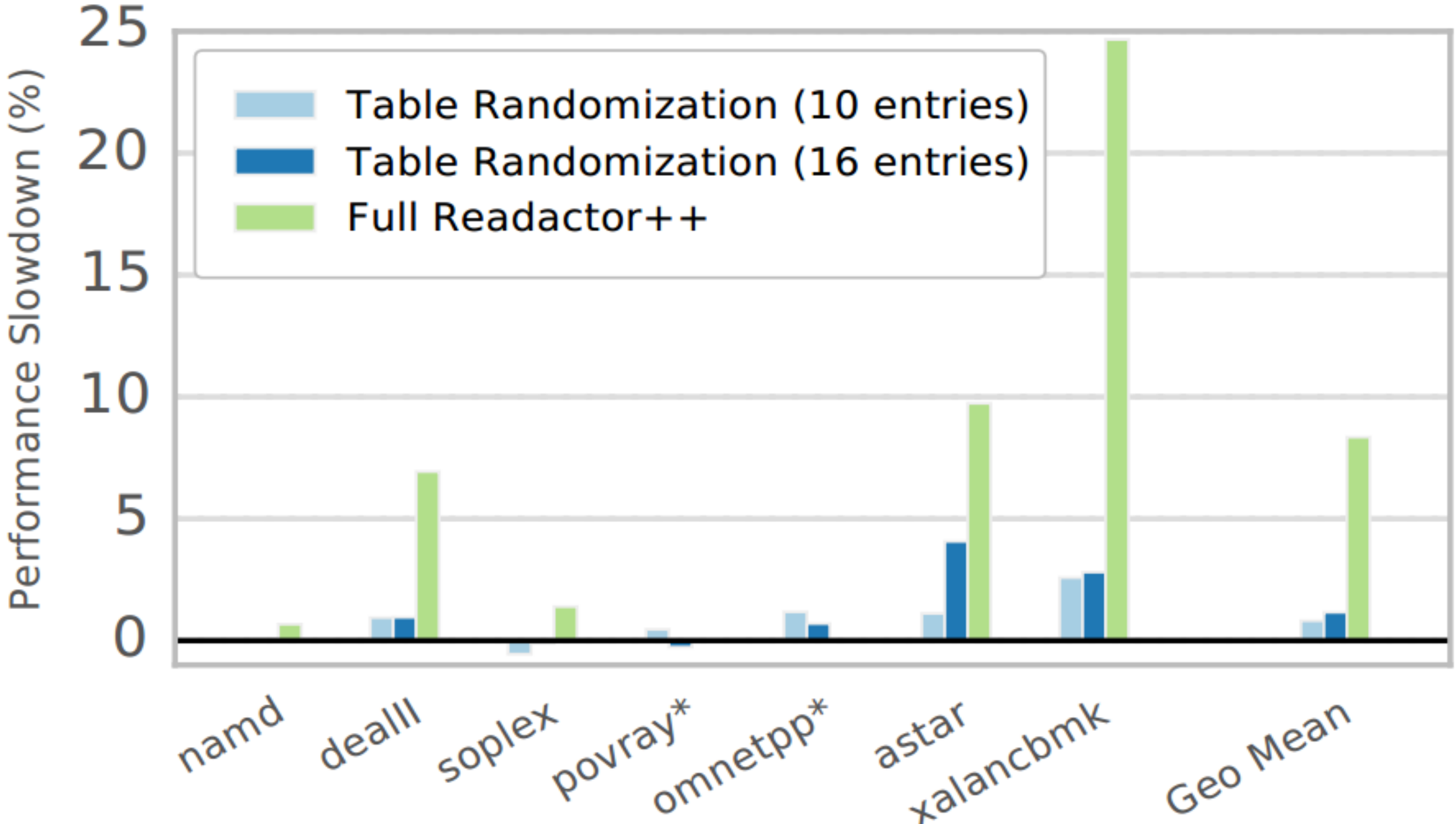
Can only measure performance of C++ components in Chromium system

Measured the performance of scrolling over a large web page

- Google
- Gmail
- CNN
- Facebook

Table randomization alone resulted in 1.0% overhead increase

The addition of x-only memory and trampolines incurred performance overhead of 7.9%


Performance Slowdown (%)
25
20
15
10
5
0
Table Randomization (10 entries)
Table Randomization (16 entries)
Full Readactor++
namd
dealII
soplex
povray*
omnetpp*
astar
xalancbmk
Geo Mean

Future Work

Other table based attacks

- C Programming Language utilizes tables of function pointers

Randomization of data structure layouts

- Further protect stack objects by randomizing their layout

Dynamically linked libraries

- Potential attack vector for clever hackers
- Libraries loaded at runtime vulnerable to COOP attack

Limitations

Readactor++ covers programs which adhere to the C++ language specification

Programs which rely on compiler-introduced changes to the VTable are not protected

VTables have different implementations between architectures

Limitations Cont.

Programs which include external libraries must be compiled with same version as external library

- External libraries may be rebuilt with Readactor++ to follow this rule

Related Work

Code Layout Randomization

- Defense schemes for obfuscating and combatting memory layout disclosure

Control-Flow Integrity

- Defend against illegal program states

Code Pointer Integrity

- Defend against code pointer exploitation

Code Layout Randomization

Oxymoron—scheme for hiding direct code references

- Oxymoron – Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing by M. Backes and S. Nurnberger
- Defends against memory disclosure by disallowing direct code references
  - Very similar to trampolines
- Vulnerable to clever use of JIT-ROP, code pages are readable

Control-Flow Integrity

COOP Safe Compiler

- Secure C++ Virtual Calls From Memory Corruption Attacks by D. Jang, Z. Tatlock, and S. Lerner
- Protect VTable pointer from modification
- Attacker is unable to point it to their malicious code
- Method is vulnerable to RILC attacks - return instructions not protected

Code-Pointer Integrity

Separate code pointers and pointers to code pointers

- Eternal War in Memory by L. Szekeres, M. Payer, T. Wei, and D. Song
- Pointers are stored in safe memory area
- Only functions deemed “trustworthy” given access to these pointers
  - Trustworthiness is determined at compile-time, not runtime
- While this scheme runs efficiently in C, programs in C++ incur a stiff overhead (>40%)

Conclusions

Code Reuse attacks (ROP, COOP) are problematic

Readactor++ is a novel defense against COOP attacks

Readactor++ has been shown to:

- Resist information disclosure
- Identify attacks and notify the host system

Using clever configuration options, Readactor++ has negligible affect on system performance

Questions?