FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

Matthias Konstantin Fischer

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

to do, title ger

Author:	Matthias Konstantin Fischer
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	M.Sc. Paul Muntean
Date:	X November, 2016

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

to do title eng

to do, title ger

Author:	Matthias Konstantin Fischer
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	M.Sc. Paul Muntean
Date:	X November, 2016

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 20. Oktober 2016 Matthias Konstantin Fischer

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

This is just an example that shows how long the abstract should be and which parts an abstract should contain.

Many applications such as the Chrome and Firefox browsers are largely implemented in C++ for its perfor- mance and modularity. Type casting, which converts one type of an object to another, plays an essential role in en- abling polymorphism in C++ because it allows a program to utilize certain general or specific implementations in the class hierarchies. However, if not correctly used, it may return unsafe and incorrectly casted values, leading to so-called bad-casting or type-confusion vulnerabili- ties. Since a bad-casted pointer violates a programmer's intended pointer semantics and enables an attacker to corrupt memory, bad-casting has critical security implica- tions similar to those of other memory corruption vulner- abilities. Despite the increasing number of bad-casting vulnerabilities, the bad-casting detection problem has not been addressed by the security community.

In this paper, we present C A V ER , a runtime bad-casting detection tool. It performs program instrumentation at compile time and uses a new runtime type tracing mechanism—the type hierarchy table—to overcome the limitation of existing approaches and efficiently verify type casting dynamically. In particular, C A V ER can be easily and automatically adopted to target applications, achieves broader detection coverage, and incurs reason- able runtime overhead. We have applied C A V ER to large- scale software including Chrome and Firefox browsers, and discovered 11 previously unknown security vulnera- bilities: nine in GNU libstdc++ and two in Firefox, all of which have been confirmed and subsequently fixed by vendors. Our evaluation showed that C A V ER imposes up to 7.6% and 64.6% overhead for performance-intensive benchmarks on the Chromium and Firefox browsers, re- spectively.

Contents

1. Introduction

Control-Flow Integrity (CFI) [11, 8] is one of the most used techniques to secure program execution flows against advanced Code-Reuse Attacks (CRAs).

Advanced CRAs such as COOP [26]

————— ROP and its manifestations (RILC, JIT-ROP, COP, COOP, JOP, Stiching Numbers ROP, Defcon 22, etc.) can be addressed with binary rewriting (user space application and OS kernel), source code recompilation or runtime monitoring such as: *(i)* fine-grained CFI with hardware support PathArmor [14], ((ii)) respectivelly coarse-grained CFI such as CCFIR [20], *(iii)* coarse-grained CFI based on binary loader CFCI [21] *(iv)* fine-grained code randomization O-CFI [18] *(v)* cryptografy with hardware support based CCFI [17] *(vi)* based on the ROP stack pivoting, PBlocker [23] *(vii)* canary based as DynaGuard [22] *(viii)* checking vTable integrity for protecting against COOP based on CFI for source code auch as SafeDispatch [25], vtv [2] LLVM and GCC compiler based vor vTable protection and binary rewriting such as vfGuard [24], vTint [19] and [16] *(ix)* with runtime hardware support based on a combination of LBR, PMU and BTS registers CFIGuard [6] *(x)* with code recompilation with CFI and/or randomization enforcement against JIT-ROP, MCFI [5], RockJIT [9] and PiCFI [10] and any combination of the above. —————-

Proposal: Name for our tool:

CCTypeMapper (Caller Calle Type Mapper)

TypeShild

TypeProtector

CCTS (caller/calle type securer or shilder)

TypeFlower

you can also make your sugestion here.

Proposal:

Citation example: [**?**].

The introduction should answer this questions:

1.What is the problem?

Specify The Problem statement.
2-3 sentences.

2.What are the current solutions?
talk about TypeArmor [15]....

3.Where the solutions lack?

4.What is your idea?

5.Contributions.

In summary, we make the following contributions:

1. We did this
2. We did this
3. We did this.

   The rest of the MA is organized as follows.

1.1. Motivation

here coms the Motivation.
2-3 sentences.

1.2. Research Goals

here coms the Motivation.
2-3 sentences.

1.3. Outline

here coms the Motivation.
2-3 sentences.
example:
The remainder of this thesis is organized as follows. Chapter 2 provides a profound background regarding VMs, VMI, and modern rootkits. We relate our work to previous research in Chapter 3. The design and architecture of WhiteRabbit is discussed in Chapter 4. This chapter comprises assumptions and necessary means that are required to meet the goals previously stated in Section 1.2. The WhiteRabbit prototype implementation is discussed and evaluated in Chapter 5 and Chapter 6, respectively. Finally, we provide an outlook concerning future work in Chapter 7 and conclude this thesis with a brief recapitulation in Chapter 8.

2. Technical Overview

2.1. Types Inference/Recovery from Binaries

2.1.1. Types Recovery in General

2.1.2. Function Parameter Types Recovery

2.1.3. Caller/Callee Parameter Types Recovery

2.2. Code-Reuse Attacks

this section will up to 1 DIN A page. First talk about code reuse attacks in general and then about COOP.

Say why COOP is not affected by the following mittigation approaches Talk about intel CET, talk about Windows CF guard.

2.3. Control-Flow Integrity

2.4. Cntrol-Flow Integrity

Part I.

Snippets

3. Snippet [AddressTaken]

As of now we used the full set of possible calltargets, which is the set of addresses of all function entry basic blocks. To further restrict the possible calltargets per callsite, we explored the notion of incorporating an address taken analysis into our application. The notion is that any indirect control flow instruction might only target addresses that are considered taken. An Address is considered to be a taken address, if it is loaded to memory or a register usually this is a constant, !optional! however it is also possible that simple calculations using multiplication and/or addition are used. We are not concerned with more complex calculations, because we have not observed compilers resorting to more complex methods and literature so far does agree [**?**].

Based on the notions of [**?**], introduced several types of indirect control flow targets of which only !shorthand! Code Pointer Constants (CK) and !shorthand! Computed Code Pointers (CC) are of interest to us. The reason for that is that the others are usually the target of indirect jumps, however we are (as of now) only interested in callsites.

Definition 3.1 *!shorthand! Code Pointer Constants (CK) are, as the name suggests, code addresses in the binary that are computed during compile time. In [**?**] it is furthermore stated that they only consider thos addresses that point to instruction boundaries or are within the range of addresses of the current module. However, we restrict that even further to only the set of addresses that point to entry blocks of functions within our current module, as these are the only valid targets of function calls.*

!optional!

***Definition** 3.2 !shorthand! Computed Code Pointers (CC) are, addresses that are computed during binary execution. In [**?**] this set only contains targets to intraprocedural indirect jumps and is thus of no interest for us*

Our approach of indentifying taken addresses is a two pronged approach. First, we iterate over the raw binary content of data segments additionally identifying possible dereferencable addresses. Second, we iterate over all instructions in functions within the disassembled binary.

3.1. step 1

We rely on Dyninst to tell us the boundaries of the .plt section !todo! add more information here and the .text section, which contain the executable part of the binary and thus are use to precheck any addresses that we might find in this step.

*As suggested in [**?**], we slide a !todo!, how much byte ? 4 or 8 ? what happens on X86-64 compared to x86window over the data sections of the binary (namely the .data, the .rodata and the .dynsym).*

3.2. step 2

*We rely on Dyninst [**?**] to supply us with the correct function bounadries and addresses of instructions, which we then pass onto our instruction decoder, which is based on DynamoRIO. In essence there are types of analysis that are performed on each instruction. First we identify all relevant constants from the instruction*

1. *If the instructions is a control flow instructions, we completely ignore it, as it cannot give us any information that is relevant. !todo! can we trace back from memory addresses and registers, what essentially is being called ?*
2. *We look a the sources and !todo! targets of the instructions and add it to the list of potentially interesting targets*
3. *If the target is a RIP-based address, we rely on DynamoRIO to decode it and also add it to the list of potentially interesting target*
4. *!todo what is with constant functions ?*
5. *!todo! can we have DynamoRIO infer the result of simple lea instructions ?*

Then for the resulting set of addresses, we check whether each either point to the entry block of a function, points within the .plt section, or is present in our reference map, which we calculated earlier. !todo! is the reference map needed ?

4. Implementation

In the Master thesis this section should be no longer than 1 DIN A page:
example of implementation text from USENIX caver Paper.
Please write in the same style.
We implemented C A V ER based on the LLVM Compiler project [43] (revision 212782, version 3.5.0). The static in- strumentation module is implemented in Clang's CodeGen module and LLVM's Instrumentation module. The runtime library is implemented using the compiler-rt module based on LLVM's Sanitizer code base. In to- tal, C A V ER is implemented in 3,540 lines of C++ code (excluding empty lines and comments). C A V ER is currently implemented for the Linux x86 platform, and there are a few platform-dependent mech- anisms. For example, the type and tracing functions for global objects are placed in the .ctors section of ELF . As these platform-dependent features can also be found in other platforms, we believe C A V ER can be ported to other platforms as well. C A V ER interposes threading functions to maintain thread contexts and hold a per-thread red- black tree for stack objects. C A V ER also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack. We also modified the front-end drivers of Clang so that users of C A V ER can easily build and secure their target applications with one extra compilation flag and linker flag, respectively.

5. Evaluation

We evaluated our tool X with Y popular servers, by instrumenting them with our tool. We performed runtime performance test with the following applications.

Our Evaluation aims to answer the following research questions:

- ***R1:*** *How efective is out tool in securing binary programs against the COOP attack?*
- ***R2:*** *How precise is our tool in detecting the types of the caller/caller pairs?*
- ***R3:*** *What is the performance overhead of our tool?*
- ***R4:*** *What are the instumentation overheads imposed by our tool*
- ***R5:*** *How many caller/called pairs are secured by our tool and how many remain unsecured?*
- ***R6:*** *Against which kind of attacks can our tool secure programs?*
- ***R7:*** *What are the Limitations of our Tool?*
- ***R8:*** *List is not exauhustive. Give another relevant research question. if there is one.*

Comparison methods. *Example: We used UBSAN (compare with TypeArmor), the state-of-art tool for detecting bad-casting bugs, as our comparison tar- get of C A V ER . Also, We used C A V ER - NAIVE , which dis- abled the two optimization techniques described in §4.4, to show their effectiveness on runtime performance opti- mization.*

Experimental setup. *Example: All experiments were run on Ubuntu 13.10 (Linux Kernel 3.11) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.*

5.1. R1: Effectiveness of our Tool

5.2. R2: Precision of our Tool

5.3. R3: Performance overhead of our Tool

5.4. R4: Instrumentation overhead of our Tool

5.5. R5: Security coverage of our tool

5.6. R6: Which kind of attacks can our tool defend off

5.7. R7: Whar are the limitations of our Tool

5.8. R8: To Do.

it is easier for the reader if we can directly map those section from underneath on the section from above.

5.9. Classification

5.9.1. Callsites

overestimation param count. table. number of parameters.

5.9.2. Calltargets

underestimation param table.

5.10. Patching Policies

Two types of diagrams. Table 5 from TypeArmor and a CDF to compare param count and param type. (baseline).

5.10.1. AT

5.10.2. ParamCount

table, cdf, baseline vs. server. approximations.

5.10.3. ParamType

table, cdf, baseline vs. server. approximations.

5.11. Security Evaluation

5.12. Performance

spec 2006.

6. Related Work

6.1. Binary-level CFI

1/2 page

6.2. Source-level CFI

1/2 page

6.3. TypeArmor Paper

1/2 page

7. Discussion

We have to define which points make sense and then talk about each other
Suggestion:

7.1. How to make the type inference more precise?

1/2 page

7.2. Comparison with TypeArmor and why are we better than TypeArmor?

1/2 page

7.3. Whys is not TypeArmor working as it should to?

1/2 page

7.4. What is not clear bout TypeArmor?

1/2 page

7.5. What can for sure not work as in TypeArmor paper explained?

Furthermore, bla ...

8. Conclusion and Future Work

This section should be no longer than 2 pages. idealy exactly 2 pages would be sufficient.

8.1. Conclusion

In this research, we presented our tool X, ...

Specify the points you want to talk about. Write 2-3 sentences about each point.

Point 1
Point 2
Point 3

8.2. Future Work

In future we bla.
Point 1
Point 2
Point 3
Specify the points you want to talk about. Write 2-3 sentences about each point.

Bibliography

[1] *M. Zhang and R. Sekar, Control Flow Integrity for COTS Binaries, In* Proceedings of the 22nd USENIX conference on Security, *(USENIX Sec), ACM, pp. 337-352, 2013.*

[2] *C. Tice et al., Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM, In* Proceedings of the 23nd USENIX conference on Security, *(USENIX Sec), ACM, 2014.*

[3] *N. Carlini et al., Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, In* Proceedings of the 24nd USENIX conference on Security, *ACM, 2015.*

[4] *R. Skowyra et al., Systematic Analysis of Defenses Against Return-Oriented Programming, In* Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses, *(RAID), 2013.*

[5] *B. Niu et al., Modular Control-Flow Integrity, In* ACM Conferece on Programming Language Design and Implementation, *(PLDI), 2014.*

[6] *R. Skowyra et al., Hardware-Assisted Fine-Grained Code-Reuse Attack Detection, In* Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses, *(RAID), 2015.*

[7] *E. Göktaş et al., Out Of Control: Overcoming Control-Flow Integrity, In* Proceedings of the 2014 IEEE Symposium on Security and Privacy, *(S&P), 2014.*

[8] *M. Abadi et al., Control Flow Integrity, In* the 12th ACM Conference on Computer and Communications Security, *(CCS), 2005.*

[9] *B. Niu et al., RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Inegrity, In* the 21th ACM Conference on Computer and Communications Security, *(CCS), 2014.*

[10] *B. Niu et al., Per-Input Control-Flow Integrity, In* the 22th ACM Conference on Computer and Communications Security, *(CCS), 2015.*

[11] *M. Abadi et al., Control Flow Integrity Principles, Implementations, and Applications, In* ACM Transactions on Information and System Security, *(TISSEC), 2009.*

[12] *N. Carlini et al., ROP is still dangerous: Breaking Modern Defenses, In* Proceedings of the 23nd USENIX conference on Security, *(USENIX Sec), ACM, 2014.*

[13] *M. Wang et al., Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries, In* Annual Computer Security Applications Conference, *(ACSAC), 2015.*

[14] *V. van der Veen et al., Practical Context-Sensiticve CFI, In* Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, *(CCS), 2015.*

[15] *Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida, A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level, In* Proceedings of the 2016 IEEE Symposium on Security and Privacy*, (S&P), 2016.*

[16] *S. Crane et al., Itś a TRaP: Table Rndomization and Protection against Function-Reuse Attacks, In* Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, (CCS), 2015.*

[17] *A. J. Mashtizadeh et al., CCFI: Cryptograhically Enforced Control Flow Integrity, In* Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, (CCS), 2015.*

[18] *V. Mohan et al., Opaque Control-Flow Integrity, In* Symposium on Network and Distributed System Security*, (NDSS), 2015.*

[19] *C. Zhang et al., VTint: Protecting Virtual Function TablesÍntegrity, In* Symposium on Network and Distributed System Security*, (NDSS), 2015.*

[20] *C. Zhang et al., Practical Control Flow Integrity & Randomization for Binary Executables, In* Proceedings of the 2013 IEEE Symposium on Security and Privacy*, (S&P), 2013.*

[21] *M. Zhang et al., Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-ROP Attcks, In* Annual Computer Security Applications Conference*, (ACSAC), 2015.*

[22] *T. Petsios et al., DynaGuard: Armoring Canary-based Protections against Brute-force Attacks, In* Annual Computer Security Applications Conference*, (ACSAC), 2015.*

[23] *A. Prakash et al., Defeating ROP Through Denial of Stack Pivot, In* Annual Computer Security Applications Conference*, (ACSAC), 2015.*

[24] *A. Prakash et al., Strict Protection for Virtual Function Calls in COTS C++ Binaries, In* Symposium on Network and Distributed System Security*, (NDSS), 2015.*

[25] *D. Jang et al., safeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks, In* Symposium on Network and Distributed System Security*, (NDSS), 2014.*

[26] *Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Counterfeit Object-oriented Programming, In* Proceedings of the 2015 IEEE Symposium on Security and Privacy*, (S&P), 2015.*