University of Windsor

School of Computer Science

COMP 8677

NETWORKING AND DATA SECURITY

DR. SHAOQUAN JIANG

REPORT

SUBMITTED BY:
KHYATI SHAH (110090234)
YESHA DESAI (110094273)
ABHINAV MALIK (110089366)

“Let Me Unwind That For You: Exceptions to Backward-Edge Protection”

This paper was written by Victor Duta, Marius Muench, Cristiano Giuffrida of Vrije Universiteit Amsterdam and Fabian Freyer, Fabio Pagani from the University of California, Santa Barbara.

It addresses the enduring issue of memory corruption vulnerabilities in programming languages like C, C++, and Objective-C. These vulnerabilities, particularly stack buffer overflow vulnerabilities, are still common despite the increased usage of safer languages. They can be exploited to hijack control flow, posing security risks.

To mitigate these vulnerabilities, strong protections such as stack canaries and shadow stacks have been implemented. These have forced attackers to develop more elaborate exploitation techniques. However, the paper argues that these protections are insufficient as they don't consider stack unwinding during exception handling, a process that depends on data located on the stack.

The authors propose a new attack paradigm called Catch Handler Oriented Programming (CHOP). CHOP can exploit the unwinding process by manipulating stack data and diverting the control flow to attacker-controlled handlers. The potential results are serious, including arbitrary code execution and arbitrary memory writes.

The paper also states that current mitigations on this attack surface are insufficient. It highlights that while the exploitation of Structured Exception Handling (SEH) is well-mitigated on Microsoft Windows, the same cannot be said for other systems. This suggests that exception handling on other systems is a largely overlooked attack surface.

The authors conducted a large-scale analysis of real-world C++ binaries to demonstrate the prevalence of exception handling semantics and assess the potential for CHOP attacks. They showed that exception usage is common in C++ software, and that CHOP offers powerful tools for exploitation. They demonstrated successful CHOP-style attacks on three real-world vulnerabilities, emphasizing that even the latest defenses (like hardware shadow stacks) fail to guard against such attacks.

I. Background

The background discusses exceptional control flow in programming and mechanisms for protecting against backward-edge control-flow hijacking. Exceptional control flow refers to how a program transitions due to faults or exceptional situations, triggering exception handling. This involves the use of exception types, throwing instances of these types, and handling exceptions using EHs (Exception Handlers). The control flow transitions between a "try" block and its associated EH.

Two main methods for protecting against backward-edge control-flow hijacking are discussed:

1. Stack Canaries: This technique, like StackGuard, places "canaries" or cookies on the stack before the return address in a function, and checks their integrity in the function epilogue before returning. This prevents attackers from manipulating the return address through stack-based buffer overflows.

2. Shadow Stacks: Shadow stacks involve saving return addresses on a separate stack that attackers can't access through stack-based buffer overflows. These addresses are used for exploit detection or protection by restoring them from the shadow stack upon return. Different variations of shadow stacks exist, with direct mapping schemes replicating the program stack's structure, while indirect schemes save only the return addresses onto the shadow stack. However, maintaining synchronization between the main stack and the shadow stack, especially during stack unwinding for exception handling, is a challenge.

We believe that exceptional control flow and control-flow hijacking are critical security concerns in programming. Exception handling is essential for dealing with unforeseen situations, ensuring robust software behaviour. Techniques like stack canaries and shadow stacks are important for preventing attackers from manipulating return addresses and executing malicious code. The discussion highlights the complexity of ensuring proper stack synchronization in the context of shadow stacks, underlining the importance of careful implementation to avoid exploitable vulnerabilities. Overall, these mechanisms play a crucial role in maintaining the security and reliability of software systems.

Exception Handling Intervals:

The exception handling discusses the internals of exception handling in programming languages, particularly focusing on stack unwinding and its implementation on Unix-based systems. This process involves runtime support through several components, including the Exception Handling ABI, personality routines, and the unwinder. The Itanium C++ ABI is a widely adopted standard for stack unwinding on Unix systems, involving a search phase and a clean-up phase. The search phase identifies appropriate exception handlers by analysing metadata associated with the current call frame, while the clean-up phase involves unwinding the stack and invoking personality routines and landing pads.

From an attacker's perspective, unwinders operate on the call stack, trusting its contents to determine which handlers to invoke. This makes call stacks an input for the unwinding state machine, and attackers who can manipulate stack data and induce exceptions can potentially exploit the unwinding process. The paper highlights potential issues with exception handling on attacker-controlled data, revealing that attacks exploiting exception handling vulnerabilities are a realistic threat.

The discussion provides valuable insights into the intricate workings of exception handling and stack unwinding, particularly emphasizing the role of runtime support and the challenges related to ensuring security and reliability in the presence of attacker-controlled data. The analysis sheds light on potential vulnerabilities that can arise when attackers manipulate exceptions and call stacks, underscoring the importance of robust exception handling mechanisms to maintain software integrity and security. The research presented here is essential for developers and security professionals to understand the complexities and potential risks associated with exception handling in modern programming environments.

Threat Model:

The threat model outlines the specific scenario and assumptions regarding potential attacks on a program's call stack due to stack corruption vulnerabilities. The main objective of the attacker is to exploit the vulnerability and gain control over the program's execution flow or memory writes for malicious purposes.

The threat model assumes the following:

1. Attacker Objective: The attacker's goal is to leverage a call stack corruption vulnerability to achieve more powerful outcomes, such as arbitrary control-flow manipulation or unauthorized memory writes. These outcomes are essential for carrying out successful exploits.

2. Attack Scope: The attacker is targeting a particular program that utilizes exception handling. The attack involves manipulating the call stack in such a way that the program throws an exception. This can occur within the vulnerable function or any callee after the stack corruption takes place.

3. Initial Stack Corruption Primitive: The attacker begins with an unconstrained stack corruption primitive. This means they can overwrite arbitrary data on the stack. The ability to control stack contents provides the foundation for crafting a more sophisticated attack.

4. Backward-Edge Mitigations: The threat model assumes the presence of backward-edge mitigations like StackGuard or shadow stacks. These techniques aim to thwart traditional exploitation methods such as Return-Oriented Programming (ROP) or return-to-libc attacks by safeguarding return addresses and preventing arbitrary code execution.

5. ASLR Bypass: While backward-edge mitigations are in place, the attacker is assumed to possess orthogonal means to bypass Address-Space Layout Randomization (ASLR). ASLR randomizes memory locations to make exploitation more difficult. The attacker's capabilities to bypass ASLR could involve various techniques, including pointer leaks, side-channel leaks, entropy exhaustion attacks, generative approaches, or partial overwrites.

This threat model provides a well-defined context for assessing the potential risks and vulnerabilities associated with stack corruption vulnerabilities in programs that involve exception handling. It acknowledges the complexities of modern security mechanisms like ASLR and backward-edge mitigations, as well as the attacker's potential capabilities to bypass them. This thorough consideration of the attacker's objectives and the existing defenses helps security professionals understand the specific challenges involved in protecting against more sophisticated attacks that leverage stack corruption vulnerabilities. It's a reminder that even with

advanced security measures in place, attackers can exploit certain scenarios to bypass these defenses, underlining the importance of a comprehensive security strategy.

II. Literature Review:

This literature review showcases the emergence and significance of Catch Handler-Oriented Programming (CHOP) attacks within the realm of cybersecurity. By combining stack corruption vulnerabilities with exception handling mechanisms, attackers demonstrate their adeptness in manipulating software behaviour.

The introduction of terminology such as "Confusion Primitives" and "Gadgets" provides a structured framework for understanding these exploits. The review delves into the capabilities of these gadgets, including control-flow hijacking and arbitrary memory manipulation.

Additionally, the exploration of diverse exploitation strategies demonstrates the depth of attackers' creativity in circumventing security measures. Overall, the review underscores the dynamic nature of cyber threats, requiring constant vigilance and adaptive defence strategies.

Here, we further study three main topics within hijacking the control flow—CHOP Attacks, the gadgets within CHOP attacks and exploitation strategies.

A. Catch Handler-Oriented Programming (CHOP) Attacks: Exploiting Stack Corruption and Exception Handling

The concept of CHOP attacks has emerged as a novel and innovative exploitation technique that combines stack corruption vulnerabilities with exception handling mechanisms. This technique allows attackers to manipulate the call stack, divert control flow, and execute their own code. The terminology associated with CHOP attacks, such as "Confusion Primitives" and "Gadgets," is introduced to provide a framework for understanding the intricate process of exploiting these vulnerabilities.

The CHOP attack process involves corrupting the saved return pointer to create confusion within the unwinding process, an integral part of exception handling. By doing so, attackers can guide the execution of attacker-

defined gadgets using controlled stack data. The section outlines various confusion primitives that attackers can employ to manipulate the unwinding process, each serving as a unique avenue for exploitation:

1. Exception Handler Landing Pad Confusion: By corrupting the saved return address and manipulating an adjacent stack frame, attackers can divert control flow to arbitrary exception handlers. This allows them to operate on attacker-controlled data.

2. Cleanup Handler Landing Pad Confusion: This technique redirects control flow to cleanup handlers before the actual exception handler is invoked. As a result, the attacker gains access to a broader range of available gadgets for exploitation.

3. SigReturn Frame Confusion: Manipulating SigReturn frames encountered during stack unwinding enables attackers to control registers and the execution flow during the unwinding process.

4. Callee-Saved Register Confusion: Corrupting callee-saved registers extends attacker control and enables CHOP attacks without the need for explicit control flow diversion.

Additionally, the section discusses how CHOP attacks transcend different regions of the virtual address space through shared libraries and their exception handling behaviours. This demonstrates the adaptability of CHOP attacks to various contexts, increasing their potential threat.

B. Gadgets within CHOP Attacks: Leveraging Exception and Cleanup Handlers

The introduction of the concept of "gadgets" within the framework of Confused Handler-Oriented Programming (CHOP) attacks sheds light on code segments exploitable by attackers, akin to traditional Return-Oriented Programming (ROP) attacks. In CHOP attacks, gadgets are embodied by exception and cleanup handlers that execute during exception handling or the unwinding process's cleanup phases. This section delves into the capabilities offered by these handlers for exploitation:

1. Backwards-edge Control-flow Hijacking: CHOP attacks provide avenues for traditional ROP and return-to-libc attacks through the manipulation of

the unwinding process. The presence of landing pad confusion in exception handlers devoid of stack canary checks enables arbitrary code execution, effectively bypassing backward-edge protections.

2. Forward-edge Control-flow Hijacking: By exploiting the ability of exception and cleanup handlers to perform indirect calls based on stack values, attackers can achieve forward-edge control-flow hijacking. This technique is especially prominent in polymorphic object destructors called via vtables.

3. Arbitrary Free: Cleanup handlers are key to invoking object destructors on the stack, which may lead to memory freeing. Attackers exploit this to achieve arbitrary memory freeing, a pivotal vulnerability leading to use-after-free exploits.

4. Arbitrary Write: Control over the stack frame of an exception handler, combined with control over callee-saved registers, results in powerful primitives. The ability to manipulate stack values and registers facilitates arbitrary write operations, potentially bypassing mitigations and enabling data-only attacks.

The exploration of gadgets within CHOP attacks underscores the sophistication of attackers in leveraging exception and cleanup handlers to attain malicious goals. The range of capabilities, spanning from arbitrary writes to control-flow hijacking, underscores the diverse attack vectors stemming from these mechanisms. This section underscores the necessity of secure coding practices, meticulous exception handling, and comprehensive security assessments for identifying and mitigating potential vulnerabilities. In an evolving threat landscape, where attackers continually explore novel techniques, defenders must remain vigilant and adapt their strategies to counter emerging cyber threats.

C. Crafting Exploitation Strategies: Primitives, Gadgets, and their Combination

The ability to craft diverse exploitation strategies emerges from the synthesis of various confusion primitives and gadget capabilities within the framework of CHOP-style attacks. These strategies exploit vulnerabilities in exception handling mechanisms to execute arbitrary code and circumvent specific security mitigations.

Golden Gadget: The concept of the "golden handler" technique within the realm of libstdc++ serves as a prime example of direct exploitation. It enables attackers to divert execution to arbitrary memory locations by manipulating the saved return address and controlling stack data. The potency lies in the indirect call facilitated by the manipulated stack data.

Pivot-to-ROP: The vulnerability classification of certain functions, housing gadgets (exception handlers), provides attackers with the opportunity to perform traditional Return-Oriented Programming (ROP) attacks. This circumvents backward-edge protections, even in the presence of stack canaries.

Data-Only Corruptions: The technique of data-only attacks emerges, wherein attackers manipulate local stack frames exclusively. By leveraging cleanup handlers or exception handlers to operate on manipulated data, attackers can avoid overwriting the return address and, in some cases, bypass the need for an Address-Space Layout Randomization (ASLR) bypass.

Cleanup-Handler Chaining: Attackers can orchestrate intricate attack sequences by corrupting a substantial portion of the stack, enabling the chaining of multiple fake stack frames with cleanup handlers. This approach resembles traditional ROP attacks, allowing for the sequential use of multiple gadgets.

SigReturn-to-ROP: The strategy of pivoting the unwinding process from the actual stack to an attacker-controlled buffer via SigReturn frame confusion stands as an exemplar. This maneuver allows for the execution of a Return-Oriented Programming (ROP) chain within the controlled buffer.

The claims elucidated in this section underscore the intricate nature of attacker techniques in manipulating exception handling mechanisms to achieve arbitrary code execution and exploit software vulnerabilities. These strategies highlight the paramount importance of robust security mechanisms and comprehensive testing to pre-empt attackers from capitalizing on system weaknesses. The section emphasizes the necessity of understanding diverse attack vectors to formulate effective defenses and underscores the ever-evolving nature of cyber threats.

III. RELATED WORK

The present paper extensively delves into the landscape of prior research and strategies concerning exception handling and unwinding attacks, offering a comparative analysis with its own novel approach. In particular, the paper underscores its divergence from prior exploits, notably those targeting the Windows Structured Exception Handling (SEH) system. It is essential to recognize that the focus of SEH-based attacks predominantly centres around the manipulation of exception handling metadata. In sharp contrast, the paper introduces and emphasizes the unique nature of Confused Handler-Oriented Programming (CHOP) attacks, which pertains to the manipulation of the data that the unwinder, a core component of the exception handling mechanism, operates upon.

By scrutinizing previous exploits and understanding their modus operandi, the paper situates its work within the broader context of vulnerability exploitation. A key distinction is outlined: while SEH-based attacks manipulate exception handling metadata, CHOP attacks pivot towards the manipulation of the data influencing the unwinding process. This distinction fundamentally shapes the characteristics and scope of the attacks, rendering CHOP attacks more versatile and adaptable in their potential application.

Moreover, the paper underscores that the uniqueness of CHOP attacks stems from their capacity to exert control over the data processed by the unwinder. This distinct approach expands the potential attack surface and grants CHOP attacks a broader applicability compared to traditional SEH-based techniques. By focusing on the data rather than the metadata, CHOP attacks can transcend the limitations of the SEH approach, potentially encompassing a wider array of software systems and scenarios.

In essence, the paper's insightful comparison between CHOP attacks and SEH-based attacks elucidates the distinct tactics employed by these two exploitation strategies. This distinction is pivotal in comprehending the novel contribution that CHOP attacks bring to the table. By illuminating the uniqueness of CHOP attacks in manipulating the unwinding process's data, the paper contributes to a deeper understanding of the evolving landscape of software vulnerability exploitation.

IV. IMPLEMENTATION AND EVALUATION

In this section we study about how attack surface and gadget analysis in the context of exception handling in C++ binaries. It provides a detailed overview of the research methodologies used to evaluate the potential attack surface in real-world software and analyze the capabilities of specific tools that an attacker might use.

For the first step of implementation the researchers collected and analysed data using a two-step process. First, they downloaded packages from the main AMD64 repository of Debian Buster and extracted the files, collecting information such as file name, associated package, file type, and original directory location. This metadata was stored in a PostgreSQL database.

The researchers then used Debian's popularity contest data, which shows the number of installations for each package, to decide which packages to prioritize for analysis. They used the Binary Ninja framework to analyse the extracted files. The results of the analysis were fed back into the database, allowing for cross-correlation of different analysis stages.

The challenge here was to accurately identify packages that handle or throw exceptions. While it might be intuitive to select packages that list libstdc++ as a dependency, the researchers found this approach to be inaccurate due to two primary reasons: (1) packages often bundle different binaries and libraries together, and a dependency relation exists as long as a single one of those requires the C++ standard library, and (2) this library is also required for binaries generated from other languages like Objective-C and Rust.

Instead, they used a more precise method where they selected only binary programs and shared libraries from the unpacked Debian packages. Then, using Binary Ninja, these programs and libraries were analyzed and flagged as programs with exception handling semantics if they either contained the .gcc_except_table section in the ELF file or called any of the libstdc++ functions to raise exceptions.

The researchers outlined two main requirements for launching a CHOP (Control-Flow Hijacking Oriented Programming) attack: (1) the ability to trigger a stack-based spatial memory corruption, and (2) the ability to throw an exception, forcing the unwinder to operate on the corrupted data.

To assess the attack surface for CHOP attacks, they developed an analysis strategy to estimate the likelihood of these primitives being present in C++ software. Stack canaries were used as a proxy for functions potentially vulnerable to memory corruption. The likelihood of an attacker being able to raise exceptions after a corruption was assessed by analyzing not only the function that can throw an exception but also any callee in the call tree with the potentially vulnerable function as the root.

The researchers also analysed the capabilities of CHOP gadgets, which provide specific corruption primitives to an attacker. They identified all the gadgets available in a binary and used static taint tracking to understand what primitives a specific gadget provides.

They assumed an attacker model where an attacker can corrupt the stack and overwrite the backward edge to confuse the unwinder, with the control limited to the callee-saved registers. Using this model, they developed a taint analysis engine based on Binary Ninja’s High-Level IL (HLIL) intermediate representation to evaluate the impact of attacker-controlled registers in these gadgets. They categorized the sinks into arbitrary free sinks, forward-edge hijacking sinks, and attacker-controlled write sinks.

Finally, each gadget was given a feasibility score, which was based on a weighted average of the number of nested branches, nested loops, basic blocks, and function calls that the taint analysis encountered before reaching the specific gadget. Lower scores indicate higher chances of reaching the gadget's sink. It's important to note that their methodology conservatively ignores potential additional attack surface from an attacker controlling the gadget's stack frame.

Several research questions based on the study's findings about exception handling in C++ software and the feasibility of CHOP attacks have been evaluated here for a possible solution. They are explained in detail below:

A. Exception Handling

Here we dive into how frequently exception handling is utilized in modern C++ software. The first approach to evaluate this was to check if a package has libstdc++ as a dependency. However, to ensure accuracy, they further refined the search. They unpacked the top 1,000 popular packages and selected all binary programs and shared libraries, which totaled to 3,303

files. Out of these, 322 binaries (approximately 9.5%) utilized exception handling. These files were then used as a dataset for the following sections. Based on these figures, they estimated that approximately 10% of programs rely on exception handling, signifying a significant number of potential targets for CHOP-style attacks.

B. Examination of Attack Scope

To understand the attack surface for CHOP attacks, the researchers evaluated the call-graph of a binary and labelled functions that can throw an exception. They used stack canaries as a proxy to estimate the functions potentially affected by stack-based memory corruption. The analysis revealed that exceptions were quite common, with 50% of the binaries having at least 40% of potentially throwing functions.

Additionally, they analysed bugs reported by OSS-Fuzz from May 2016 to June 2022. The dataset included 38,464 bugs reported across 344 software projects written in C++. After sifting through the bug reports, they selected 442 stack-related bugs found in C++ applications that could potentially be exploited using CHOP.

C. Analysis of Gadgets

To assess the power of CHOP gadgets available to an attacker, the study used taint analysis on Debian binaries. The results were categorized into cleanup handlers and catch handlers, detailing their capacity to permit arbitrary free or write-what gadgets. While cleanup handlers often allowed arbitrary free, write-what gadgets were frequently missing in both handler categories. However, forward-edge control-flow hijacks were found common in both categories. This indicates that about 90% of analyzed binaries contained at least one of the mentioned gadget types, suggesting a high likelihood of exploitation through CHOP-based techniques.

The study also noticed a correlation between the size of the binaries and the number of gadgets. On average, a 1% increase in the size of a binary led to an approximate increase of 1.1% in the number of gadgets.

It is important here to note that the evaluation of the applicability of CHOP to real-world software, the platforms affected beyond Linux, and the effectiveness of recent mitigations against CHOP are not studied here by

the authors of the paper and the variables accounting to them are out of scope.

V. Case Study

The paper delves into a new form of attack vector known as Control-flow Hijack Oriented Programming (CHOP). This unique cybersecurity threat takes advantage of software bugs that are not effectively detected or mitigated by traditional security measures such as StackGuard. The paper carries out an in-depth exploration of CHOP attacks by detailing three case studies that highlight these vulnerabilities in real-world scenarios.

In the first case study (CVE-2009-4009), a vulnerability in the PowerDNS recursor is targeted. It is a straightforward example, where the bug allows for a buffer overflow and then raising a runtime_error exception. The attacker initiates a DNS query that causes the vulnerable binary to request an attacker-controlled server, which responds with the exploit payload. The payload overflows the stack buffer, overwrites the stack canary and the saved return address, and sets the return address to a "golden gadget" in libstdc++, resulting in a forward-edge control-flow hijacking.

In the second case study (CVE-2018-5809), a vulnerability in LibRaw, a popular image processing library, is exploited. The attack here is more complicated as the exception is thrown only under specific conditions and requires manipulation of stack data. The payload overflows the stack buffer and changes the saved return address to a pivot-to-ROP gadget, triggering an exception that leads to the unwinding of the previously corrupted stack.

The third case study (SCSSU-201801 or CVE-2018-4300) involves the Common Access Card (CAC) module of smartcardservices running on macOS Sierra. Here, the certificate may be compressed, and if decompression fails, an exception is thrown, causing the unwinder to operate on attacker-controlled data. This exploit shows that CHOP attacks can be effective on platforms other than Linux.

The paper claims that existing mitigation strategies are insufficient to protect against CHOP attacks. It suggests that hardening measures for unwinding logic are needed, which I agree with. Traditional countermeasures like StackGuard fail to protect against such attacks, and

the paper also demonstrates that CHOP attacks can work on different platforms, broadening their potential impact.

However, the paper's findings are based on the assumption that ASLR is broken, either via a single pointer leak or other bypassing techniques. While the authors were able to exploit the vulnerabilities within two days of development, in real-world scenarios, the attacker might not always have such an advantage. So, while I agree with the paper's overall assertion about the potential danger of CHOP attacks, the practical application of these attacks may be more difficult than it seems.

As for a programming experiment, it could be insightful to conduct a similar analysis on more recent vulnerabilities and see if CHOP could exploit them. An interesting experiment would be to try and develop a mitigation strategy for the unwinding logic and test its effectiveness against a simulated CHOP attack. However, it's important to make sure that any such experiments are performed in a controlled and ethical manner, without causing harm or potential security risks.

VI. Exploitability for additional Platforms and Mitigations

The section you provided focuses on two aspects: the exploitability of the test program across different platforms, and the efficacy of various mitigation strategies.

Researchers tested the program across seven configurations, encompassing Linux, macOS, Windows, Android, and iOS, utilizing both x86_64 and AArch64 architectures. The findings revealed that the control flow of the application could be effectively altered on all tested systems.

Next, they investigated if shadow stacks, a recent safeguard technique designed to ensure backward-edge control flow integrity, could resist these attacks. They evaluated various shadow stack implementations, such as six schemes outlined in Burow et al.'s SoK on shadow stacks, LLVM's ShadowCallStack tailored for AArch64 on Android, and Intel CET on Windows. However, all tested implementations fell short in preventing the attack.

From these results, the researchers concluded that the existing unwinder implementations and backward-edge protection mechanisms fall short in

protecting against the attacks introduced in their study. The only resilient method discovered was LLVM's SafeStack, which uses a unique split stack design to implement return address isolation. Yet, even SafeStack has its limitations, as it can't detect attacks, nor can it prevent return address corruption caused by more sophisticated corruption methods. In addition, SafeStack's incompatibility with dynamic libraries limits its effectiveness in defending against exploits rooted in dynamic library vulnerabilities.

Finally, they recommend the use of forward-edge protection mechanisms to mitigate such exploits. However, this approach can only guard against direct forward-edge control-flow hijacks, leaving systems susceptible to backward-edge hijacks, data-only attacks, and general arbitrary writes.

The paper first introduces immediate mitigation strategies against Catch Handler Oriented Programming (CHOP)-style attacks. These attacks involve corrupting a program's exception handling to take control of its execution. An immediate defensive approach proposed is to broaden the scope of stack canary checks to the stack unwinding path. Stack canaries are a security measure to prevent stack buffer overflow attacks. During the stack unwinding process, the program state is reset to handle an exception. Extending canary checks here would help in detecting corruption at throw time and inhibit the attacker from executing malicious logic.

However, the full implementation of this strategy would require every function that calls __cxa_throw to be guarded by a stack canary. The authors argue that while this would require updates to compiler toolchains, the performance cost should be low since unwinding only occurs under exceptional circumstances.

For cases where the attacker can bypass stack canaries, advanced hardening strategies are proposed. These include the introduction of a context-aware unwinding mechanism or making modifications to the unwinder so that it operates on shadow stack data instead of the original stack data. The authors note that these modifications would involve significant changes and potentially conflict with the Itanium C++ ABI, a specification for exception handling in C++ on Itanium and other platforms.

The paper also proposes minimizing the usage of 'catch-all' and other lenient catch handlers, which could serve as opportunities for an attacker. Instead, using specific handlers for each exception and reducing the

number of handlers in widely-used libraries could greatly limit an attacker's capabilities.

Randomization-based defenses, such as function reordering and stack layout randomization, are also suggested as measures to help protect against CHOP-style attacks. However, the authors note that these might not be effective against advanced versions of these attacks.

VII. DISCUSSION AND CONCLUSION

The authors discuss a number of topics in this section. They talk about the potential issues that could arise when trying to return to normal program control flow after executing the attacker's code, suggesting potential mitigation methods like using a golden handler or restoring the program state via a Return Oriented Programming (ROP) chain.

They also discuss the usage of taint analysis, which helps assess the feasibility of exploiting a CHOP-style vulnerability. However, the authors note that while this analysis is useful, it does not provide automated exploit generation capabilities.

The authors then talk about the applicability of their research to languages other than C++. They argue that their findings are likely relevant to any language that uses stack unwinding during exception handling, which includes languages like Objective-C and Rust.

Finally, they acknowledge potential threats to the validity of their research, such as the fact that their dataset is derived from the Debian package index and does not include proprietary software. However, they argue that the attack surface for CHOP is likely comparable in both free and non-free software.

In conclusion, the authors argue that the process of stack unwinding during exception handling is an attack surface that has been overlooked in the past. They show through their research that current protections are not enough against CHOP-style attacks, providing evidence for the need for more robust security checks in production unwinding software.