d t A logo of a university of windsor

Description automatically generated with low confidence

|  |
| --- |
| **Course Name** |
| COMP8677-3-R-2023F|Networking and Data Security (Monday) |

|  |
| --- |
| **Document Type** |
| Report on Research Paper |

|  |
| --- |
| **Paper Title** |
| Let Me Unwind That For You: Exceptions to  Backward-Edge Protection |
| **Research Paper Authors** |
| Victor Duta, Fabian Freyer, Fabio Pagani  , Marius Muench and Cristiano Giuffrida |

|  |
| --- |
| **Professor** |
| Dr. Shaoquan Jiang |

|  |
| --- |
| **Graduate Assistant** |
| Ms. Salma Aly |

|  |
| --- |
| **Group Number & Group Name** |
| MonG 2 |

|  |
| --- |
| **Team - Members                               Student ID** |
| Manjinder Singh                                 110097177 |
| Harbhajan Singh                                 110100089 |
| Karan Vishavjit                                   110099867 |

|  |
| --- |
| **Submitted On** |
| December 9th, 2023 |

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

1. **Introduction**

This research paper, authored by Victor Duta, Marius Muench, Cristiano Giuffrida from Vrije Universiteit Amsterdam, and Fabian Freyer, Fabio Pagani from the University of California, Santa Barbara, investigates the persistent challenge posed by memory corruption vulnerabilities in programming languages like C, C++, and Objective-C. Despite the growing adoption of safer languages, vulnerabilities, particularly stack buffer overflow vulnerabilities, remain prevalent. These vulnerabilities pose significant security risks as they can be exploited to take control of the program's flow. Efforts to address this issue have led to the implementation of robust protections, such as stack canaries and shadow stacks, which in turn prompt attackers to develop more sophisticated exploitation techniques. However, the paper argues that existing protections fall short by neglecting the consideration of stack unwinding during exception handling—a process intricately linked to data on the stack.

To address these inadequacies, the authors propose a novel attack paradigm called Catch Handler Oriented Programming (CHOP). CHOP exploits the stack unwinding process by manipulating stack data, redirecting control flow to attacker-controlled handlers. The potential consequences are severe, encompassing arbitrary code execution and unrestricted memory writes. The paper highlights the insufficiency of current mitigations on this attack surface. Despite the effective mitigation of Structured Exception Handling (SEH) exploitation on Microsoft Windows, the authors emphasize a critical oversight in other systems. This suggests that exception handling on alternative systems represents a largely overlooked attack surface.

To support their claims, the authors conducted an extensive analysis of real-world C++ binaries, showcasing the prevalence of exception-handling semantics and evaluating susceptibility to CHOP attacks. Their findings reveal the widespread use of exception handling in C++ software, emphasizing that CHOP provides potent tools for exploitation. The authors further demonstrated successful CHOP-style attacks on three real-world vulnerabilities, highlighting the failure of even the latest defenses, such as hardware shadow stacks, to safeguard against such exploits.

1. **Background**

Addressing exceptional control flow in programming involves managing transitions caused by faults or exceptional situations, leading to the activation of exception handling mechanisms. This encompasses the utilization of various exception types, the throwing of instances corresponding to these types, and the orchestration of exception handlers (EHs). The control flow dynamically shifts between a "try" block and it associated EH. Two primary strategies for safeguarding against backward-edge control-flow hijacking have been explored.

Firstly, the implementation of Stack Canaries, such as in StackGuard, introduces "canaries" or cookies on the stack preceding the return address within a function. Subsequently, the integrity of these canaries is verified in the function epilogue before execution returns. This method acts as a deterrent against attackers manipulating the return address through stack-based buffer overflows.

Secondly, the adoption of Shadow Stacks involves the preservation of return addresses on a distinct stack inaccessible to attackers during stack-based buffer overflows. These addresses contribute to exploit detection or protection by restoration from the shadow stack upon return. Various iterations of shadow stacks exist, with direct mapping schemes replicating the program stack's structure and indirect schemes exclusively saving return addresses on the shadow stack. Nevertheless, maintaining synchronization between the main stack and the shadow stack, especially during stack unwinding for exception handling, poses a significant challenge.

It is imperative to recognize that exceptional control flow and control-flow hijacking present pivotal security concerns in the realm of programming. Exception handling emerges as a critical component for managing unforeseen situations and ensuring the resilient behavior of software. Techniques like stack canaries and shadow stacks play a pivotal role in thwarting attempts by attackers to manipulate return addresses and execute malicious code. The discourse underscores the intricacy of ensuring proper stack synchronization within the context of shadow stacks, underscoring the significance of meticulous implementation to mitigate exploitable vulnerabilities. In essence, these mechanisms assume a vital role in upholding the security and reliability of software systems.

1. **Exception Handling Intervals**

Exploring the intricacies of exception handling within programming languages, particularly delving into the nuances of stack unwinding and its implementation on Unix-based systems, forms the crux of this examination. The runtime support for this process involves various components, such as the Exception Handling ABI, personality routines, and the unwinder. Notably, the Itanium C++ ABI stands out as the widely accepted standard for stack unwinding on Unix systems, encompassing both a search phase and a clean-up phase. The search phase discerns suitable exception handlers by scrutinizing metadata linked to the current call frame. Conversely, the clean-up phase involves the unwinding of the stack, triggering personality routines and landing pads.

Examining the scenario from a potential attacker's viewpoint, it becomes evident that unwinders rely on the call stack's contents to ascertain which handlers to invoke. This design inherently positions call stacks as inputs for the unwinding state machine. Consequently, attackers armed with the capability to manipulate stack data and induce exceptions may exploit the unwinding process to their advantage. The paper underscores the tangible threat posed by attacks exploiting vulnerabilities in exception handling, shedding light on potential issues when dealing with attacker-controlled data.

The discourse offers invaluable insights into the intricate mechanics of exception handling and stack unwinding, placing particular emphasis on the role of runtime support and the challenges associated with upholding security and reliability in the face of attacker-controlled data. The analysis elucidates potential vulnerabilities that may arise when attackers manipulate exceptions and call stacks, underscoring the critical importance of robust exception handling mechanisms to preserve software integrity and security. This research proves indispensable for developers and security professionals seeking to comprehend the complexities and potential risks inherent in exception handling within contemporary programming environments.

1. **Threat Model**

A comprehensive understanding of the threat model is paramount, as it delineates the specific scenario and assumptions surrounding potential attacks on a program's call stack attributable to stack corruption vulnerabilities. In this context, the primary objective of the attacker centers around exploiting the identified vulnerability to gain control over the program's execution flow or manipulate memory writes for malicious purposes. This model has following assumptions:

1. **Backward-Edge Mitigations:**

The assumption here is that security measures like StackGuard or shadow stacks are implemented to thwart conventional Return-Oriented Programming (ROP) or return-to-libc attacks. These techniques focus on securing the return addresses in the call stack, making it harder for attackers to manipulate program flow.

1. **Address-Space Layout Randomization (ASLR) Bypass:**

While assuming the presence of backward-edge mitigations, the paper acknowledges the possibility of attackers having orthogonal means to bypass Address-Space Layout Randomization (ASLR). This entails scenarios where attackers can overcome the randomized placement of key program elements in memory. The potential bypass methods include traditional pointer leaks, side-channel leaks, entropy exhaustion attacks, generative approaches, or partial overwrites.

1. **Orthogonal Means to Bypass ASLR:**

The assumption involves acknowledging that the attacker possesses alternative methods to overcome Address-Space Layout Randomization (ASLR). These methods include traditional pointer leaks, where attackers obtain memory address information through non-randomized channels, side-channel leaks leveraging unintended information leakage, entropy exhaustion attacks depleting the randomness of ASLR, generative approaches crafting predictable layouts, or partial overwrites manipulating specific memory regions.

This threat model designed for threat assessment establishes a systematic framework to evaluate potential risks and vulnerabilities associated with instances of stack corruption in programs that incorporate exception handling. It considers the complexities introduced by contemporary security measures, including Address Space Layout Randomization (ASLR) and protections against backward-edge exploits. Simultaneously, it recognizes the potential of attackers to bypass these defenses. The thorough analysis of the attacker's objectives and existing protective measures assists security professionals in understanding the specific challenges involved in safeguarding against more advanced attacks that exploit vulnerabilities related to stack corruption. This comprehensive examination serves as a reminder that, despite the presence of sophisticated security mechanisms, ongoing vigilance is essential to address the ever-evolving landscape of threats.

1. **Literature Review**

This literature review explores the increasing significance of Catch Handler-Oriented Programming (CHOP) attacks within the realm of cybersecurity. These attacks effectively exploit vulnerabilities related to stack corruption in conjunction with exception handling mechanisms, showcasing attackers' adeptness at manipulating software behaviour.

In this paper, authors introduced two terms. The introduction of these terms such as "Confusion Primitives" and "Gadgets" provides a structured framework for understanding these exploits. The review thoroughly investigates the capabilities of these gadgets, encompassing aspects like control-flow hijacking and arbitrary memory manipulation. Additionally, it examines various exploitation strategies, shedding light on the attackers' creativity in circumventing security measures. In essence, the review underscores the dynamic nature of cyber threats, emphasizing the continuous need for vigilance and adaptive defense strategies.

In the following sections, we will delve deeper into three key aspects related to control flow hijacking: CHOP Attacks, the sophisticated gadgets employed in CHOP attacks, and the varied range of exploitation strategies utilized by attackers.

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

In the realm of cybersecurity, a novel exploitation technique called CHOP attacks has surfaced, seamlessly merging stack corruption vulnerabilities with exception handling mechanisms. This innovative approach enables attackers to manipulate the call stack, redirect control flow, and execute malicious code. To understand the intricacies of CHOP attacks, we introduce terms such as "Confusion Primitives" and "Gadgets," forming a conceptual framework for comprehending the exploitation process. This section delves into the CHOP attack process, elucidating the corruption of the saved return pointer to induce confusion in the unwinding process—an integral aspect of exception handling. By doing so, attackers gain the ability to guide the execution of attacker-defined gadgets using controlled stack data. The section further explores various confusion primitives that attackers can employ, each providing a distinct avenue for exploitation:

1. **Exception Handler Landing Pad Confusion:**

By manipulating the saved return address and influencing an adjacent stack frame, attackers can divert control flow to arbitrary exception handlers. This manipulation grants them access to operate on attacker-controlled data. Attackers exploit this confusion to execute malicious code with precision, compromising system integrity.

1. **Cleanup Handler Landing Pad Confusion:**

This technique redirects control flow to cleanup handlers before the invocation of the actual exception handler. Attackers gain access to a broader range of available gadgets for exploitation. By leveraging this confusion primitive, attackers position themselves strategically within the exception handling process, maximizing the potential for executing malicious actions while maintaining a covert presence.

1. **SigReturn Frame Confusion:**

The manipulation of SigReturn frames encountered during stack unwinding empowers attackers to control registers and the execution flow during this unwinding process. Expanding on this confusion primitive, attackers can orchestrate intricate attacks by carefully manipulating SigReturn frames, allowing for precise control over the system's state and facilitating more sophisticated exploits.

**d. Callee-Saved Register Confusion:**

Corrupting callee-saved registers extends attacker control and facilitates CHOP attacks without explicit control flow diversion. By corrupting these registers, attackers gain control over the immediate execution path and establish a foothold for more prolonged and sophisticated attacks, potentially compromising the system's overall security posture.

Additionally, the section explores the transcendent nature of CHOP attacks across various regions of the virtual address space, particularly emphasizing their adaptability to different contexts through shared libraries and their respective exception handling behaviours. This adaptability underscores the versatile and potent threat posed by CHOP attacks.

1. **Overview of CHOP Attacks and Gadgets in Exception and Cleanup Handlers:**

The concept of "gadgets" within the realm of Confused Handler-Oriented Programming (CHOP) attacks sheds light on exploitable code segments, reminiscent of traditional Return-Oriented Programming (ROP) attacks. Gadgets in CHOP attacks emerge as exception and cleanup handlers that execute during the handling of exceptions or the cleanup phases of unwinding processes. This section aims to delve into the capabilities inherent in these handlers for potential exploitation:

1. **Control-flow Manipulation via Backwards-edge:**

CHOP attacks introduce opportunities for traditional ROP and return-to-libc attacks by manipulating the unwinding process. The presence of landing pad confusion in exception handlers lacking stack canary checks allows for arbitrary code execution, effectively bypassing protections against backward-edge attacks. This underscores the importance of robust stack protection mechanisms to thwart such manipulations.

1. **Control-flow Manipulation via Forward-edge:**

Leveraging the ability of exception and cleanup handlers to execute indirect calls based on stack values empowers attackers to achieve control-flow hijacking in the forward-edge. This technique is particularly noteworthy in polymorphic object destructors invoked through vtables. Defenders should focus on secure design patterns for vtables and implement runtime integrity checks to mitigate this form of attack effectively.

1. **Exploiting Cleanup Handlers for Unrestricted Freeing:**

Cleanup handlers play a pivotal role in invoking object destructors on the stack, potentially leading to memory freeing. Attackers exploit this functionality to accomplish arbitrary memory freeing, a critical vulnerability that can result in use-after-free exploits. Implementing stringent memory management practices and ensuring proper object lifecycle management can mitigate the risk associated with these exploits.

1. **Exploiting Exception Handlers for Unrestricted Write:**

Gaining control over the stack frame of an exception handler, coupled with control over callee-saved registers, provides potent primitives. The ability to manipulate stack values and registers facilitates arbitrary write operations, potentially circumventing mitigations and enabling data-only attacks. Organizations should enforce strict control over exception handling mechanisms and conduct thorough code audits to identify and rectify potential vulnerabilities.

The scrutiny of gadgets within CHOP attacks underscores the adeptness of attackers in leveraging exception and cleanup handlers to achieve malicious objectives. The diverse range of capabilities, spanning from arbitrary writes to control-flow hijacking, underscores the various attack vectors stemming from these mechanisms. This section highlights the significance of secure coding practices, meticulous exception handling, and comprehensive security assessments to identify and mitigate potential vulnerabilities. In an ever-evolving threat landscape, where attackers continuously explore new techniques, defenders must remain vigilant and adapt their strategies to counter emerging cyber threats.

1. **Devising Strategies for Exploitation: Primitives, Gadgets, and its Combination:**

Formulating a variety of exploitation strategies entails combining different elements of confusion and leveraging tool capabilities within the CHOP-style attack framework. These strategies are designed to exploit vulnerabilities in exception handling mechanisms, allowing the execution of arbitrary code and the circumvention of specific security safeguards. The adept amalgamation of confusion primitives and gadgets serves as the foundation for these strategic exploits.

1. **Golden Gadget:**

An illustrative instance of direct exploitation involves the "golden handler" concept within the context of libstdc++. This technique empowers attackers to redirect 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, showcasing the skillful manipulation of execution flow for malicious purposes.

1. **Pivot to ROP:**

Certain functions, housing gadgets like exception handlers, offer attackers the opportunity to employ traditional Return-Oriented Programming (ROP) attacks. This method effectively bypasses backward-edge protections, even in the presence of stack canaries. Identifying and exploiting vulnerable functions in exception handling opens avenues for evading critical security measures.

1. **Data-Only Corruptions:**

Data-only attacks come into play when attackers exclusively manipulate local stack frames. By utilizing cleanup handlers or exception handlers to process manipulated data, attackers can sidestep overwriting the return address and, in some cases, eliminate the need for an Address-Space Layout Randomization (ASLR) bypass. This nuanced approach to manipulating stack frames demonstrates a sophisticated evasion of conventional security measures.

1. **Cleanup Handlers Chaining:**

Complex attack sequences can be orchestrated by corrupting a significant portion of the stack, enabling the chaining of multiple fabricated stack frames with cleanup handlers. This approach resembles traditional ROP attacks, allowing for the sequential use of multiple gadgets. The orchestration of cleanup-handler chains showcases a higher level of intricacy in executing exploit sequences.

1. **SigReturn-to-ROP:**

Shifting the unwinding process from the actual stack to an attacker-controlled buffer via SigReturn frame confusion is an exemplary strategy. This makeover facilitates the execution of a Return-Oriented Programming (ROP) chain within the controlled buffer. The SigReturn-to-ROP strategy underscores the exploitation of control flow mechanisms in a manner that evades typical security measures.

The statements presented in this section illuminate the nuanced techniques employed by attackers in manipulating exception handling mechanisms to achieve arbitrary code execution and exploit software vulnerabilities. These strategies underscore the critical importance of robust security measures and comprehensive testing to prevent attackers from exploiting system weaknesses. The section underscores the need to comprehend diverse attack vectors to formulate effective defenses, highlighting the ever-evolving nature of cyber threats.

1. **Related Works**

The current study extensively delves into the realm of previous research and strategies related to handling exceptions and thwarting attacks, offering a comparative analysis alongside its own innovative approach. Notably, the paper underscores its departure from prior exploits, specifically those targeting the Windows Structured Exception Handling (SEH) system. In this context, the authors highlight the need to explore alternative avenues beyond conventional SEH-based strategies for more effective security measures.

It is crucial to highlight that SEH-based attacks primarily involve manipulating exception handling metadata. In contrast, this paper introduces and emphasizes the distinctiveness of Confused Handler-Oriented Programming (CHOP) attacks, which revolve around manipulating the data processed by the unwinder—a core component of the exception handling mechanism. By elucidating the limitations of relying solely on metadata manipulation, the paper advocates for a paradigm shift in vulnerability exploitation strategies, emphasizing the importance of considering the underlying data manipulation in security frameworks.

By conducting a thorough examination of past exploits and gaining an understanding of their methodologies, the paper contextualizes its work within the broader landscape of vulnerability exploitation. A significant point of differentiation is stressed: while SEH-based attacks manipulate exception handling metadata, CHOP attacks focus on manipulating the data influencing the unwinding process. This critical analysis not only underscores the need for diverse approaches in the security domain but also positions CHOP attacks as a more nuanced and adaptive strategy for addressing evolving cyber threats.

Moreover, the paper underscores that the uniqueness of CHOP attacks stems from their capacity to control the data processed by the unwinder. This distinctive approach broadens the potential attack surface, providing CHOP attacks with a wider applicability compared to traditional SEH-based techniques. In recognizing the broader implications of data-focused manipulation, the paper advocates for a holistic understanding of software vulnerability exploitation, encouraging researchers and practitioners to explore the intricacies of data control as a crucial aspect of future security frameworks.

Essentially, 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 vital for understanding the novel contribution that CHOP attacks bring to the forefront. By shedding light on the uniqueness of CHOP attacks in manipulating the data of the unwinding process, the paper contributes to a deeper comprehension of the evolving landscape of software vulnerability exploitation, paving the way for more robust and comprehensive security measures.

1. **Implementation & Evaluation**
2. **Dataset Collection:**

Script is setup to crawl and download package metadata from Debian repositories and target the main AMD64 repository. script is designed to handle a significant amount of data, specifically 51,625 package metadata entries. Once the data is downloaded, the next step is to extract each package. During this extraction process, it's crucial to record specific file metadata, such as the file name and path, and then save this information in a database PostgreSQL in this case. This approach ensures to have a detailed record of all the packages and their contents.

After the extraction and documentation phase, integrate these packages into the Binary Ninja pipeline. (Binary Ninja is a tool that allows for deeper analysis of the binary files within these packages).

Once the analysis through Binary Ninja is complete, the results are fed back into the database. This integration ensures that the database contains both the raw metadata of the files and the insights from the Binary Ninja analysis.

1. **Identifying binaries:**

To analyse a subset of Debian exception handling packages, Researchers started by unpacking 1,000 priority packages from the repository. The goal is to narrow down to specific files, and filter to isolate only the executable binaries and shared libraries. This process resulted in a set of 3,303 files and these binaries were disassembled using Binary Ninja.

Goal is to identify binaries that use exception handling, to achieve this, researchers looked for the presence of a '.gcc\_except\_table' section or calls to 'cxa\_throw', which are indicative of exception handling usage in the code. Through this targeted search, resulted in 322 binaries that included exception handling features.

1. **Attack Surface analysis:**

Next is to analyse that attack surface by looking at each binary and find any functions that call "\_stack\_chk\_fail". This indicates that the function has implemented stack canaries, a security mechanism designed to prevent stack buffer overflow attacks. Then recursively trace call graphs up to 7 levels within each binary.

Then examine each function identified in these traces to check if it calls the 'cxa\_throw'. The presence of this call is significant as it denotes functions capable of throwing an exception.

Finally, to assess security risks, estimate the attack surface of these binaries by correlating the findings of functions with stack canaries and those capable of throwing exceptions.

1. **Gadget Analysis:**

In detailed analysis of exception handlers present in the binaries, parse the .eh\_frame and .gcc\_except sections. This is crucial to extract metadata about the exception handlers. then associate these handlers with their respective function boundaries by utilizing call-site data, which helped in accurately pinpointing the start and end of each function.

Next, for every identified exception handler, research employed Binary Ninja for static taint tracking. This is used to follow the flow of data through the binary, particularly focusing on how it interacts with sensitive operations or functions, known as taint sinks. A taint sink is essentially a critical point in the code, like a memory write or code execution operation, that could be exploited by tainted data.

The analysis involved a meticulous sink detection process, where research specifically looked for taint sinks that could be indicative of vulnerabilities, such as those enabling memory writes or code execution. If tainted data reached a sink is identified, it signals that the handler could potentially enable malicious capabilities.

Then rank and filter these 'gadgets' or code snippets based on the feasibility of reaching these sinks. This provide with an estimate of how exploitable each gadget might be. By focusing on these aspects, researchers were able to assess the security implications of exception handling in the binaries more effectively, highlighting potential vulnerabilities and areas requiring closer scrutiny.

**Evaluation:**

The research aims to answer below question:

**Q: How prevalent is the usage of exception handling in modern software?**

Around 10% of debian packages uses c++ exception handling. This shows that exception handling is prevalent enough to provide a significant attack surface if vulnerabilities exist in throwing functions.

**Q: How large is the attack surface for CHOP attacks?**

The analysis showed that 35% of throwing functions operate on stack buffers. This indicates a high number of functions vulnerable to buffer overflows that can allow corruption of control data like return addresses. Functions that throw exceptions after potential memory corruptions provide a prime attack avenue.

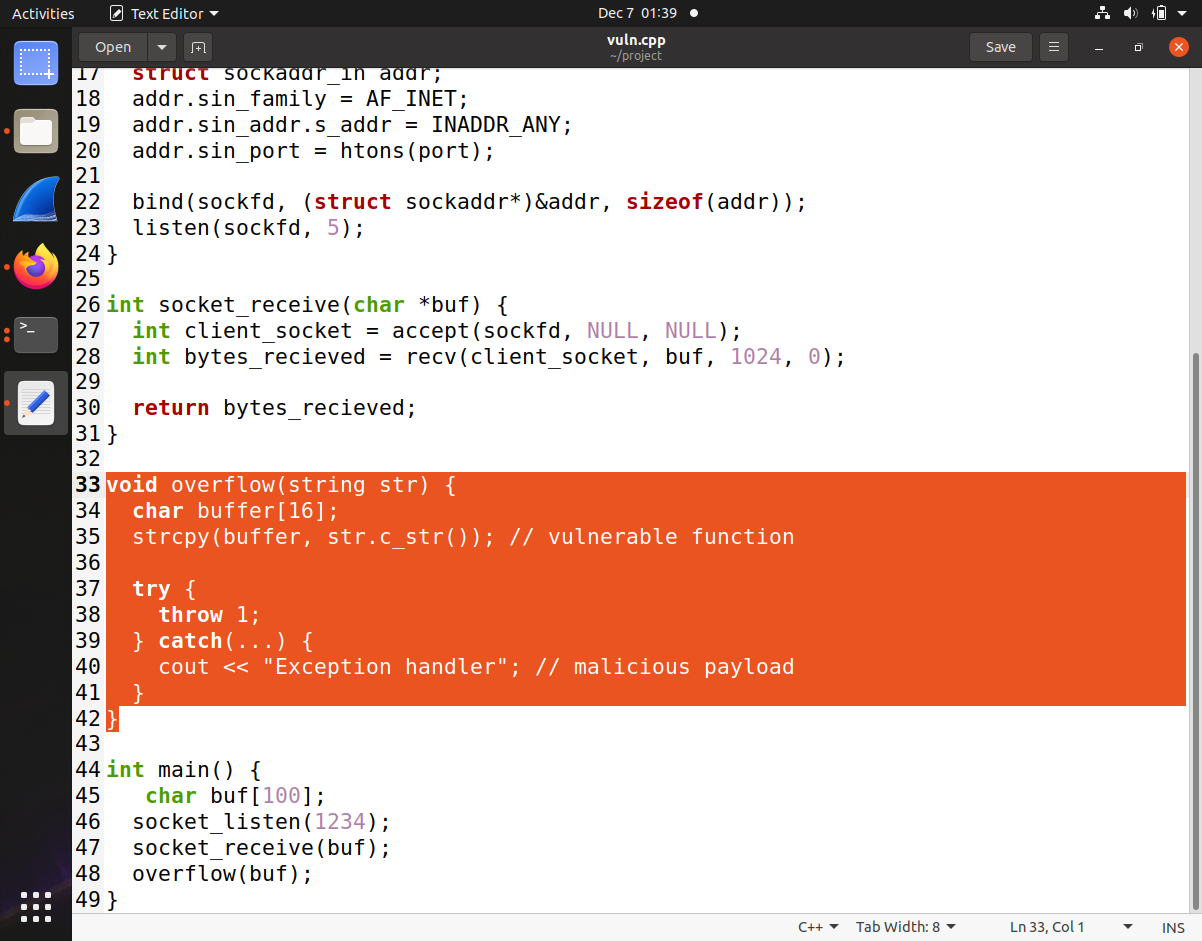
**Q: How powerful are the CHOP gadgets available to an Attacker?**

The evaluation demonstrated powerful exploit primitives enabled via exception handler diversion. 79% binaries had arbitrary free gadgets while over 90% had some critical exploit gadget. 36% also had forward control flow hijack gadgets. By diverting execution to exception handlers, attackers can leverage these preexisting code snippets to achieve malicious effects like arbitrary writes or code execution during the unwinding process.

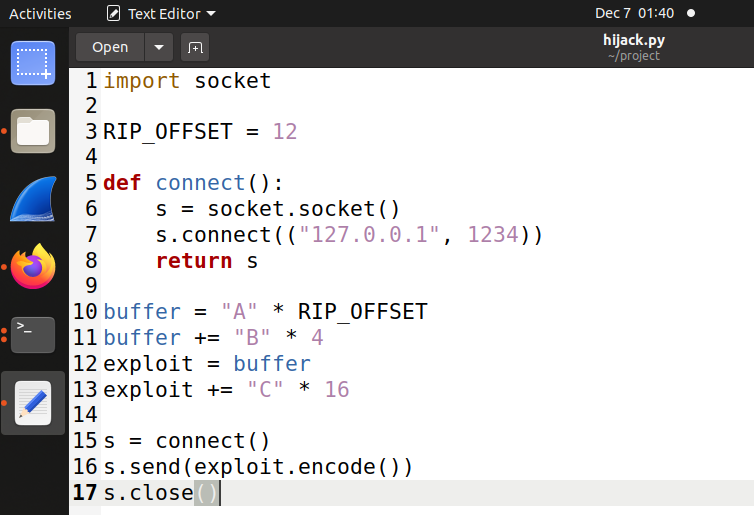
**Q: Are recent mitigations effective against CHOP?**

Shadow stacks are a mitigation technique that protects the integrity of return addresses. However, testing showed that they still fail to prevent attacks that divert code flow to exception handlers for exploitation. Across operating systems and architectures, the analysis highlighted that bypassing shadow stacks protections is still possible using exception handler diversion. So additional defensive mechanisms are needed.

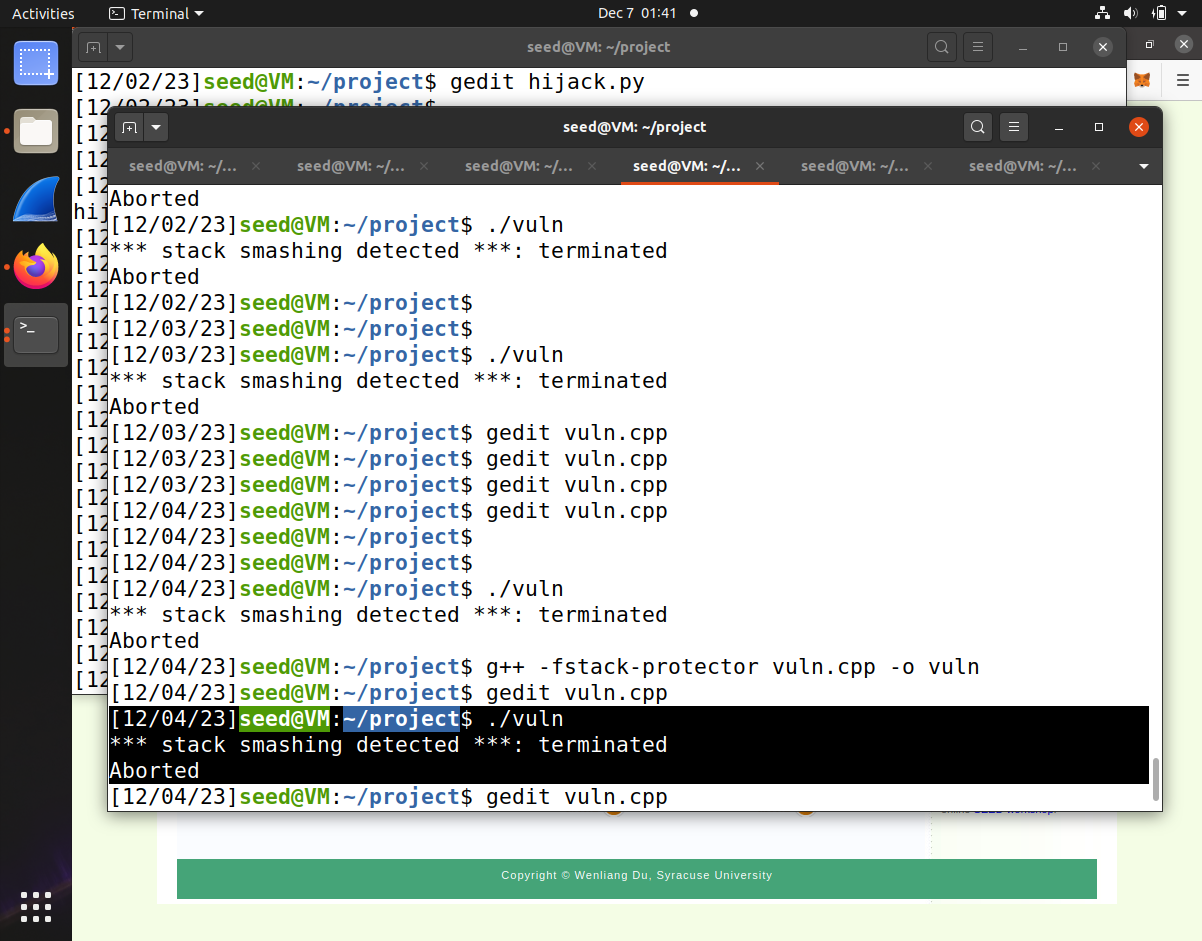
**Demonstration**:



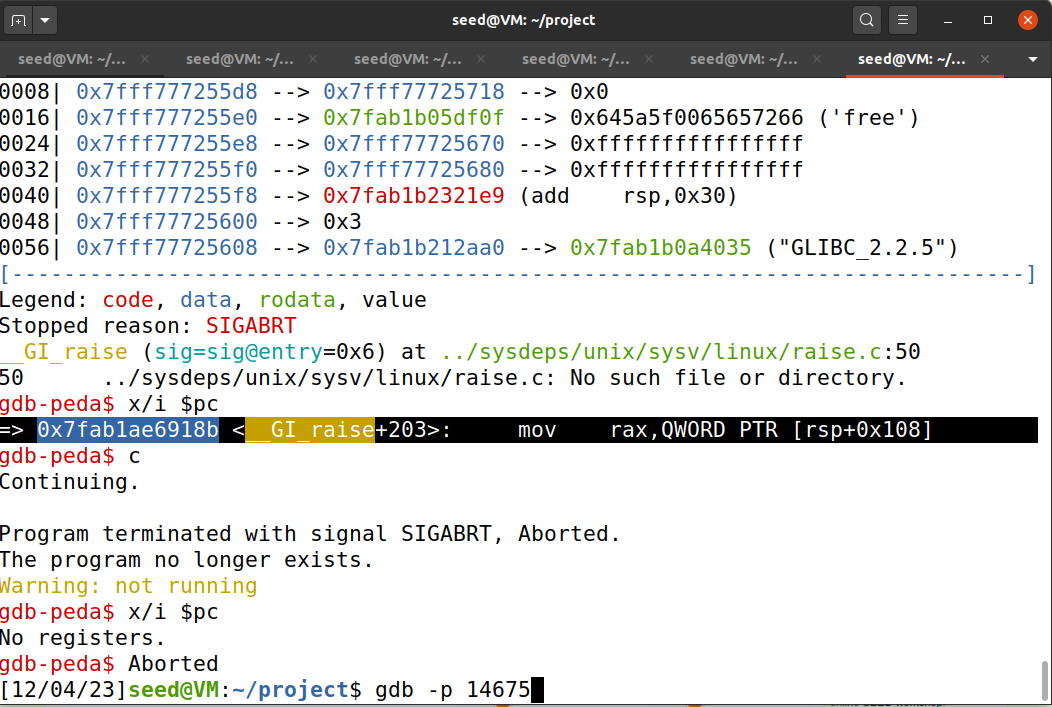
The vulnerable C++ program contains a buffer overflow in the overflow () function, which copies user input from a socket to a 16-byte stack buffer using strcpy(). This overflows the buffer, corrupting adjacent stack memory. After the overflow, overflow () throws a generic exception, which will trigger stack unwinding after memory has already been corrupted.



The Python exploit script sends a payload that includes buffer overflow strings designed to overwrite the saved return address with four B's. Later, it triggers the exception by sending a dummy string.



When examined in gdb, the program receives a SIGABRT abnormal termination signal, indicating an error occurred.



The instruction pointer register RIP shows it is about to execute the \_\_GI\_raise function to handle the raised exception 2 (SIGABRT). The stack pointer register RSP points to the corrupted buffer. Further examination shows the program counter is about to execute a mov instruction that reads from the corrupted stack area.

In summary, the attack has successfully corrupted the stack, including overwriting the saved return address, caused an exception to be thrown which triggers unwinding, and diverted control flow to the \_\_GI\_raise exception handler. This handler is now going to operate on the corrupted stack data with malicious values injected by the script. This demonstrates control flow diversion to an exception handler after corrupting stack state.

1. Case Study

**Real World Case Studies:**

1. **PowerDNS Recursor:** Debina lenny package of PowerDNS running on Ubuntu 20.04.4 system has vulnerable code in “questionExpand” function used for DNS query processing used a fixed-size stack which can be overflowed.

**Exploit Method:** The overflow occurred when parsing DNS responses. The researchers sent a crafted DNS response that exceeded the buffer size the code explicitly checks whether more data has been written into the buffer than it can hold and throws a runtime error the check happens only after data is written in next iteration and then triggered a runtime error, exploiting the overflow.

**Outcome:** The overflow and subsequent exception handling allowed the researchers to overwrite the saved return address with a pointer to a "golden gadget" in the libstdc++, leading to control flow hijacking and execution of a ROP (Return-Oriented Programming) chain, ultimately opening a remote shell for the attacker.

1. **LibRaw:** an image processing library having buffer overflow vulnerability in “parse\_exif” function when parsing images with a specific EXIF tag.

**Exploit Method:** The exploit involved modifying maker note of an existing image created by Rasberry Pie camera to trigger the buffer overflow and overwrite the saved return address. The overflow was extended to change the camera model data, triggering a different function (parse\_makernote) that threw an exception, leading to stack unwinding.

**Outcome:** The researchers crafted a second MakerNote tag to trigger the exception, which, combined with the corrupted stack, led to the execution of a ROP chain and allowed the attacker to execute arbitrary commands which eventually opened shell.

1. **Smart-card Services on macOS:** Common Access Card (CAC) module of smart-card services running on macOS Sierra has a vulnerability related to certificate data being read into a stack buffer.

**Exploit Method:** The researchers used an error condition in the certificate decompression process to trigger an exception. They used the Virtual Smart Card Architecture with modifications to deliver the exploit payload.

**Outcome:** The payload overwrote the saved return address with a pointer to a pivot-to-ROP gadget. The ROP chain then executed commands specified by the attacker on the victim system, demonstrating CHOP's applicability beyond Linux platforms.

**Mitigation Strategies:**

**Extending Stack Canaries** Every function that calls \_\_cxa\_throw to throw exceptions should be protected by a stack canary, not just exception throwing functions. Additionally, functions that handle exceptions should also incorporate stack canary protections. This extends protection coverage to unwinding paths invoked during exception handling.

**Context-Aware Unwinding** The unwinder can store information about legitimate registered exception handlers. It can initiate the unwinding process only if an appropriate handler for the thrown exception type is registered. Alternatively, in the presence of shadow stacks, the unwinder can be modified to operate on shadow stack data instead of the original stack.

**Limiting Overly Permissive Catch Handlers** The usage of generic, catch-all exception handlers should be limited, encouraging exception types to be handled by specific handlers instead. However, using third-party libraries with exception handling can still expose programs to control hijacking via exception handling attacks.

**Randomization Defenses** like function reordering in binaries and altering stack memory layouts can introduce randomness to impede attacks. However, fine-grained code randomization approaches may not be incompatible with C++ exception handling assumptions and metadata. Also, advanced attack techniques like data-oriented programming may bypass randomization.

1. **Discussions & Conclusion**

In this section, the authors thoroughly explore various subjects, delving into the intricacies of addressing potential challenges linked to restoring normal program control flow after an attacker's code execution. They propose practical mitigation strategies, such as implementing a golden handler or reinstating the program state through a Return Oriented Programming (ROP) chain. Moreover, they discuss the nuances of these strategies, underscoring their significance in the evolving landscape of cybersecurity defenses. Additionally, the authors shed light on the dynamic nature of these challenges, emphasizing the need for adaptive solutions that can withstand evolving attack methodologies.

Furthermore, the authors engage in a comprehensive discussion on the broader applicability of their research, arguing that their findings extend beyond the confines of C++. They convincingly assert that programming languages utilizing stack unwinding during exception handling, such as Objective-C and Rust, are likely susceptible to similar vulnerabilities. Expanding on this, they delve into specific examples and draw parallels, demonstrating the universality of their identified attack surface. To address potential criticisms regarding the scope of their dataset, the authors acknowledge its limitation to the Debian package index, yet they present a compelling case for its representative nature across both free and non-free software.

In summary, the authors posit that the often-overlooked aspect of stack unwinding during exception handling constitutes a significant attack surface. Their research not only highlights the inadequacies of prevailing protections against CHOP-style attacks but also advocates for a paradigm shift toward more adaptive and comprehensive security checks in production unwinding software. The evolving threat landscape, as illuminated by the authors, necessitates a continuous and dynamic reassessment of defensive strategies to effectively mitigate emerging risks in software security.