Exploiting Common Code Vulnerabilities

Alexandre Jerónimo[1†] and Gonçalo Paulino[1†]

[1]School of Technology and Management, Polytechnic Institute of Leiria, Campus 2, Morro do Lena – Alto do Vieiro, Apartado 4163, 2411-901, Leiria, Portugal.

Contributing authors: 2230467@my.ipleiria.pt; 2230469@my.ipleiria.pt;
[†]These authors contributed equally to this work.

Abstract

This article presents a comprehensive security analysis of the custom-made "dnsProxy" server, addressing vulnerabilities in code injection, buffer overflow, privilege escalation, and Denial of Service (DoS). The study involves practical scenarios where a controlled environment is established, simulating real-world threats. To fortify the dnsProxy server against potential attacks, preventive measures are proposed, including rate limiting and the implementation of the Token Bucket Algorithm.

Keywords: cybersecurity, common vulnerabilities, security analysis, malware, code injection, buffer overflow, privilege escalation, Denial of Service

1 Introduction

In the dynamic field of cybersecurity, recognising and comprehending potential vulnerabilities play a pivotal role in enhancing digital defences. This project delves into core aspects of secure software development — specifically, vulnerabilities and exploits. Through a focused examination and implementation of selected vulnerabilities, including classic issues like code injection, buffer overflow, and privilege escalation, this effort seeks to deepen our understanding of these security challenges. The objective is to provide a nuanced exploration that can inform and guide our knowledge in addressing these security risks effectively.

This project revolves around the development of a malware-type application, specifically designed to exploit common vulnerabilities found in software code. It serves as an educational exercise, providing insights into the mechanisms of cyber threats and,

equally importantly, the preventive measures that can be employed to mitigate these risks.

Before delving into the practical implementation and exploration of these vulnerabilities, this report begins with a theoretical foundation. The subsequent sections will detail the nuances of each vulnerability, providing definitions and concepts, illustrative code examples, and discussions on common solutions.

2 Examined Vulnerabilities

In this section, we explore several vulnerabilities that are essential to secure software development. As mentioned previously, our analysis focuses on code injection, buffer overflow, privilege escalation, and denial of service. Each vulnerability is examined with a focus on clarity, offering definitions and practical code examples to help understand their nuanced intricacies.

2.1 Code Injection

A Code Injection Vulnerability exists when an attacker is able to insert their code into the program's execution flow. This vulnerability is usually enabled by the lack of validation and/or sanitisation of user input [1].

SQL Injection (SQLi), Command Injection, and Cross-Site Scripting (XSS) are three of the most popular instances of code injection. These instances may be exploited to perform a wide variety of attacks, such as phishing, compromising sensitive information, denial of service, gaining unauthorised access to the system, privilege escalation, and more. The specific types of attacks that a particular vulnerability might enable are limited by the program function in which they reside.

The best way to prevent code injection vulnerabilities is by validating and correctly sanitising the user input when required. It is also important to thoroughly test an application's user inputs with tools like Burp Suite.

Code Examples

Let's imagine a scenario where a user with malicious intentions seeks unauthorised access to an admin dashboard. In this attempt, they exploit an SQL Injection vulnerability by inserting `OR '1'='1'; --` into the username field.

```php
<?php // Assuming $username is taken from user input
$username = $_POST['username']; // Assuming $_POST['username'] = "' OR '1'='1; --'"
$password = $_POST['password'];
// Vulnerable SQL query
$query = "SELECT * FROM admins WHERE username = '$username' AND password = '$password'";
```

```sql
-- Resulting SQL query:
SELECT * FROM admins WHERE username = '' OR '1'='1'; --' AND password = '[...]'
```

Fig. 1: Example code vulnerable to SQL injection

In Fig. 1, we can observe an instasnce of PHP code susceptible to SQLi. The code lacks proper sanitisation of the user-inserted username and password. By setting the

username to `' OR '1'='1'; --`, we effectively alter the resulting SQL expression to the one displayed in the figure. This expression, instead of looking for one with a particular username and password, lists every user from the admin table functionally logging the attacker in as the first user in the results.

```php
<?php
// Assuming $username is taken from user input
$username = $_POST['username']; // Assuming $_POST['username'] = "' OR '1'='1; --'"
$password = $_POST['password'];
// Use prepared statements to prevent SQL injection
$query = $conn->prepare("SELECT * FROM users WHERE username = :username AND password = :password");
$query->bindParam(':username', $username);
$query->bindParam(':password', $password);
```

```sql
-- Resulting SQL query:
SELECT * FROM admins WHERE username = '\' OR \'1\'=\'1; --\'' AND password = '[...]'
```

Fig. 2: Patched SQL injection vulnerability

In Fig. 2 we can observe one of the many possible solutions that fix this vulnerability. In this case, we make use of the MySQL prepared statements to add a layer of protection by separating the user input from the SQL code. The `bindParam()` method automatically sanitises the user input, preventing a potential SQLi by parsing the username as a literal string instead of part of the command. The resulting username in this example would be `\' OR \'1\'=\'1\'; --`.

2.2 Buffer Overflow

A buffer overflow is a software vulnerability that occurs when a program writes more data to a buffer, or temporary data storage area, than it was allocated for. A buffer overflow vulnerability can be exploited to overwrite data adjacent to the buffer, ultimately resulting in the alteration of the program's normal behaviour [2]. The severity of this vulnerability differs on a case by case basis, depending on how the program's code enabled it and the manner in which it is exploited, to the extent that two buffer overflows in the same program might exhibit completely different behaviours. Exploiting a buffer overflow in one instance might not at all hinder the program's code execution flow, whilst in another one it might have harsher consequences, such as a privilege escalation or crashing the application, potentially provoking a form of a Denial of Service attack.

To protect against this type of vulnerability, it is important to implement good coding practises when handling string and memory manipulation related code/functions. In addition to general strategies that should be implemented across all vulnerability types — e.g., regularly reviewing and auditing a program's code to ensure its safety — there are more specific measures that can be applied to specifically address buffer overflow vulnerabilities. Whenever possible, using alternative functions specifically developed to automatically verify that the data being written into a buffer does not exceed the allocated space is advisable. It is also recommended to enable memory protection mechanisms in both the program's compiler and the operating system it runs on.

Code Examples

In Fig. 3, we can observe code that is vulnerable to a buffer overflow. The usage of the unsafe `strcpy()` function and the code's improper memory manipulation causes the contents of the `allowAccess` variable to be overwritten whilst altering the value of `buffer`. This happens because the value of `allowAccess` is stored in memory right after the space allocated for `buffer` (4 bytes, 1 for each `char`). When writing 7 bytes to `buffer`, its contents along with the next 3 bytes of memory are overwritten; therefore, in this case, `allowAccess` changes from its original value of 0 to 53 (the decimal value of `char` "5"). Due to this change, the user is then unexpectedly given access to a privileged feature of the program.

```c
// Strings in C are null terminated, so the buffer must be 1 character longer than the string
char buffer[4];        // 0xff010
char allowAccess = 0;  // 0xff014

// Vulnerable strcpy function without proper bounds checking
strcpy(buffer, "123456");  // "123456" is 7 characters long, but buffer only accepts 4 characters

if (allowAccess) printf("Access granted!\n");
```

Fig. 3: Example code vulnerable to a buffer overflow

In Fig. 4 we can observe the patched code that fixes the buffer overflow presented on the previous figure. Here, instead of the `strcpy()` method, we employ `strncpy()`, enabling us to specify the number of characters it should copy. By setting that number to the size of the buffer, we guarantee that no more than 4 characters (including the null terminator) are written to the buffer.

```c
char buffer[4];

// strncpy is a safer alternative to strcpy, as it enables
// the specification of the number of characters to copy
strncpy(buffer, "123456", sizeof(buffer));

printf("Buffer: %s\n", buffer);  // This will print "123"
```

Fig. 4: Patched buffer overflow vulnerability

2.3 Privilege Escalation

Privilege Escalation is a security concern that arises when an attacker gains unauthorised access to higher levels of system privileges through programming bugs, design flaws or configuration errors. The vulnerability allows an assailant to exploit elevated permissions beyond their originally designated access level [3].

There are two types of privilege escalation:

- Horizontal privilege escalation, which occurs when an attacker with limited access attempts to gain unauthorised access to the account of another user with similar privileges [4].

- Vertical privilege escalation, which involves an attacker escalating their privileges to a higher level, gaining access to functionalities reserved for users with more extensive permissions [4]. This is the type of privilege escalation we will focus on.

Preventing privilege escalation involves implementing robust access controls and ensuring that user permissions are strictly defined and adhered to. Regular security audits, user permission reviews, and monitoring for suspicious activities are essential components of a comprehensive security strategy.

Mitigating privilege escalation involves implementing robust access controls and ensuring that user permissions are strictly defined and adhered to. Given that each type of privilege escalation exploit varies, the solutions must be tailored on a case-by-case basis. In this context, regular security audits, user permission reviews, and vigilant monitoring for suspicious activities are indispensable components of a comprehensive security strategy.

Avoiding privilege escalation vulnerabilities demands a proactive strategy, encompassing the following key principles: Adhering to the Principle of Least Privilege (PoLP) ensures that user privileges align strictly with their designated roles and responsibilities. Regular audits, involving periodic reviews of user permissions, serve to identify and rectify any discrepancies that may arise. Additionally, the implementation of real-time monitoring enables swift detection and response to potential threats, reinforcing the overall security posture against privilege escalation exploits [4].

2.4 Denial of Service

Denial of Service (DoS) is a security concern that revolves around disrupting the availability of a system or network, rendering it temporarily or indefinitely inaccessible to legitimate users. This vulnerability is exploited by overwhelming the target with an excessive volume of requests, causing a depletion of resources or rendering the system unresponsive [5].

DoS vulnerabilities can have severe consequences, including service unavailability, loss of revenue, and damage to the reputation of the affected entity. Attackers may employ various techniques, such as Distributed Denial of Service (DDoS), to amplify the impact and complicate mitigation efforts.

Effectively preventing DoS vulnerabilities requires a thorough strategy. This includes the implementation of traffic filtering mechanisms to identify and block malicious traffic, the application of rate limiting measures to restrict the number of requests from a single source and prevent resource exhaustion, and the utilisation of redundancy and load balancing techniques [6]. The latter ensures the distribution of traffic across multiple servers, thereby ensuring system availability, even when confronted with a targeted attack.

3 Practical Scenario

In the practical implementation of our project, we established a controlled environment to simulate a real-world scenario. We selected Debian 12 “bookworm” as the operating system, where we created two distinct user accounts, each assigned specific roles:

- The “debian” account, with root privileges, facilitated system-level installations and configurations, and was used to install Docker.
- The “dnsproxy” account, without root privileges but belonging to the “docker” group, swhich gives it permission to use Docker for deploying containerised applications.

We then installed the “dnsProxy” program and ran it as a service, serving as the thematic server for this practical exploration. The program’s service was configured to run under the user account “dnsproxy.”

4 The “dnsProxy” Program

We intentionally developed the “dnsProxy” program to encompass the vulnerabilities outlined in this article, explicitly with the aim of exploiting them. This client-server application functions as a DNS proxy, forwarding domain names provided by clients to a server. The server, in turn, utilises the `dig` command to resolve these domain names using external DNS servers, returning the corresponding IPv4 address. All connections between clients and the server are established through TCP sockets.

We started by developing the standard functionality for both the dnsProxy server and client applications. Following this, we intentionally modified the code of the server application to incorporate the vulnerabilities we intended to examine. Subsequently, we integrated the necessary code into the client application to automatically exploit these vulnerabilities. Additionally, we implemented a menu that offers the option to choose between the normal functionality and the exploitation of each programmed vulnerability. This menu includes the following options: TCP Socket Connection, Simple Code Injection Menu, Buffer Overflow Attack, Hacking Scenario and DoS Attack.

The following subsections will explain each of the added vulnerabilities along with a proposed solution for each of them.

4.1 Code Injection

The first vulnerability we added to the server application was a code injection vulnerability that allowed the client to send a domain name with shell commands attached to it. By injecting these commands into the domain, the attacker can trick the server into executing unintended and potentially malicious instructions.

To enable this, we purposefully neglected the sanitisation of the domain name before resolving it using `dig`. The line of code responsible for building the `dig` command is the following:

```
sprintf(command, "dig @1.1.1.1 +short +timeout=5 +retry=0 %s", domain);
```

Due to the absence of domain validation, an attacker can exploit this code vulnerability effortlessly by setting the domain name to `. &&` followed by the command they intend to execute on the server (e.g., `. && echo test`). The resulting command executed by the hacker could resemble the following:

```
dig @1.1.1.1 +short +timeout=5 +retry=0 . && echo test
```

Seeing that the `dig` command, with the given parameters, produces no output when attempting to resolve the domain name “.” (dot), if the output of the injected command is small enough, the server might even reply with part of it. This is due to the server being programmed to read up to 20 bytes from the `dig` output.

4.1.1 With Space Protection

Afterwards, we simulated a rudimentary solution that an inexperienced software developer might attempt upon discovering this vulnerability. This endeavor aims to highlight the significance of adopting an attacker's mindset when addressing vulnerabilities or handling security matters in general. The basic solution involved parsing the received domain name only up to the first blank space. However, this approach proved insufficient, as an attacker could replace blank spaces in their payload with the `${IFS}`[1] variable, effectively achieving the same results as before (e.g., `.&&echo${IFS}test${IFS}with${IFS}spaces`).

4.1.2 Solution

In both described code injection scenarios, the solution is straightforward and centres around proper input validation for the domain supplied to the server by the clients. An illustrative example is depicted in Fig. 5, showcasing a validation approach that ensures the domain name comprises only alphanumeric characters, periods, and hyphens.

```
1  // Check if the domain is valid
2  int isValid = 1;
3  for (size_t i = 0; i < strlen(domain); i++)
4      if (!isalnum(domain[i]) && domain[i] != '.' && domain[i] != '-')
5          isValid = 0; // Domain is invalid...
```

Fig. 5: Proper Validation of the Domain Name

4.2 Buffer Overflow

We introduced a buffer overflow as the second vulnerability into the server application's code. This flaw enables an attacker to circumvent the existing security validation within the program. More specifically, by exploiting this vulnerability, one could potentially bypass the solution implemented for the previous security issue discussed in the earlier subsection. Consequently, this could open the door once again to a code injection exploit.

The issue stems from the mismanagement of allocated memory space and the use of the less secure `strcpy()` function. As depicted on Fig. 6, the code begins by declaring two variables and allocating space to them. While this may seem fine initially, the oversight lies in failing to account for the additional space required in the `command` variable to store the contents of the command's prefix, along with the maximum possible size for the domain string[2]. Given that the command's prefix is 39 characters long, the `command` variable should have 294 characters of space allocated to it.

The predicament arises when using `strcpy()`; in a scenario where the client sends a domain name with the maximum possible size, it will overwrite 39 additional spaces of memory. Since `validDomain` was declared immediately after `command`, there's a

[1] A special variable that contains a blank space.
[2] 253 characters + null terminator = 254 characters

high likelihood that it will be overwritten from its original value after completing its validation.

```
char *command = malloc(sizeof(char) * 254);  // The correct value should be 294
uint8_t *validDomain = malloc(sizeof(uint8_t) * 1);
(*validDomain) = 1;

// Check if the domain is valid
for (size_t i = 0; i < strlen(domain); i++)
    if (!isalnum(domain[i]) && domain[i] != '.' && domain[i] != '-')
      (*validDomain) = 0;

sprintf(command, "dig @1.1.1.1 +short +timeout=5 +retry=0 %s", domain);
if ((*validDomain) == 0) cancel(); // Domain is invalid, cancel execution
```

Fig. 6: Server code vulnerable to a buffer overflow vulnerability

In the client application, we provide an option that exploits this vulnerability by prompting the user for the command they wish to execute on the server's system. We then pad it with additional characters to reach the required size for the buffer overflow, effectively overwriting the `validDomain` variable. Before incorporating those extra characters, we insert a hash sign ("#") to prevent the bash shell from attempting to interpret them.

Solution

Addressing this vulnerability is straightforward: allocate the appropriate amount of memory space for the variable and opt for memory-safe alternatives like `strncpy()`.

4.3 Denial of Service

Regarding Denial of Service (DoS) vulnerabilities, the dnsProxy server initially lacked any implemented mitigations. To demonstrate the impact of this vulnerability, we devised a DoS attack strategy. In this attack, the client initiates a multitude of threads. Each thread in this attack attempts to connect to the server and promptly disconnects immediately upon successful connection. This process repeats in an infinite loop until the user manually halts the execution of the client.

The rapid and incessant connect-disconnect cycle implemented in this DoS attack can overwhelm the server, resulting in a denial of service to legitimate users and eventually leading to a server crash. This illustrative scenario highlights the significance of implementing robust DoS protections to fortify the server against such disruptive attacks.

In response to this threat, we implemented two rudimentary examples of DoS preventive measures.

Preventive Measure: Rate Limiting

We implemented a rudimentary rate limiting system as the first preventive measure. This system performs checks before accepting a new connection, verifying whether either the maximum number of requests per minute or the maximum number of

concurrent connections has been reached. If either limit is reached, the server ignores the new connection request.

The per-minute rate limit functions by employing two separate counters, one runs from the start of the minute to the end, and the other from half of one minute to half of the next. This is done to address a *blind spot* in the rate limit. For instance, if there is a maximum of 100 connections per minute and 99 connections occur in the last second of a minute, with another 99 connections in the first second of the next minute, there would, in fact, be more than 100 connections per minute; however, the first counter would be reset to zero at the start of the new minute, resulting in the rate limit not triggering.

These rate limiting mechanisms serve as a protective barrier, preventing the server from being overwhelmed by excessive connection requests and enhancing its resilience against potential DoS attacks.

Preventive Measure: Token Bucket

As a complementary measure, we implemented the Token Bucket Algorithm. This algorithm operates by managing a virtual "bucket" of tokens. With each incoming request, a token is depleted from the bucket. If the bucket is empty, the request is either denied or delayed. In cases where the rate limit is exceeded, the system enforces a delay by pausing the request for one second. This proactive approach also helps to regulate the incoming traffic, preventing rapid and excessive requests that could otherwise overwhelm the server.

5 Simulation of Exploitation

In addition to the aforementioned work, we devised a "Hacking Scenario" to simulate the actions a malicious actor might take when exploiting these vulnerabilities. By selecting the "Hacking Scenario" option in the client application's main menu and providing the program with an SSH public key, it exploits the buffer overflow vulnerability mentioned in section 4.2 and executes the necessary commands on the server's system to allow a SSH connection to it. This results in the disclosure of the username of the account that dnsProxy runs on, and the addition of the SSH key to that account's `~/.ssh/authorized_keys` file, which replaces the need for a password.

5.1 Privilege Escalation

Upon accessing the system's `dnsproxy` account, the intruder quickly realises that this account lacks superuser privileges. This limitation sets the stage for the privilege escalation vulnerability intentionally introduced into the system.

Despite lacking superuser privileges, the `dnsproxy` account has permission to use Docker, due to being included in the `docker` group. This poses a significant threat, as Docker access offers a relatively straightforward route to gaining root access, including superuser privileges. To exploit this vulnerability, we developed a bash script designed to be executed from an account with Docker access. Upon execution, the script creates a new account granted with superuser privileges and outputs the credentials to the terminal. The potential ramifications are severe, as a malicious actor could use these

credentials to SSH into the new account, thereby gaining complete control over the machine.

The script is designed to create a new user with superuser privileges on a system, utilising an Alpine Docker container. It initiates the process by generating a random 12-character alphanumeric password. Subsequently, the script executes the container, mounting the root filesystem at `/mnt` within it. Employing `chroot`, the root directory is changed to `mnt`, providing access to the host's filesystem. Within this context, the script creates a new user with the specified username and password, incorporates the user into the sudo (superuser) group, and exits the container. Finally, the script displays the username and password in the terminal, concluding by performing clean-up actions on the container, dangling images, and build cache, with the intention of eliminating any potential traces.

Preventive Measures

To effectively mitigate the privilege escalation vulnerability linked to Docker, a key preventive strategy involves configuring it to run the Docker daemon as a non-root user — commonly referred to as "rootless mode" — whenever feasible. Rootless Docker operates without needing superuser privileges, providing an additional layer of protection against potential vulnerabilities in both the Docker daemon and container runtime [7].

In scenarios where rootless mode is not applicable, it becomes imperative to prudently restrict Docker access to specific accounts. This precautionary measure limits the number of accounts with access to Docker, reducing the potential surface area for security threats and helping mitigate the risk associated with unauthorised usage.

6 MBR Overwriter

As an additional aspect of our project, we ventured into the development of a malicious program designed to replace the Master Boot Record (MBR) of a boot device, thereby preventing any subsequent boot-ups. This involved thorough research to understand the complexities of developing a custom MBR. Valuable resources found on a GitHub repository [8] provided the foundation for developing an Assembly code-based MBR. Using NASM — an assembler for the x86 CPU architecture — we compiled the Assembly code into binary code (machine code). The final touch involved creating a C program capable of deploying our custom payload by overwriting the first 512 bytes of the boot device.

The successful execution of this process resulted in the virtual machine becoming unbootable, displaying a custom message of our choosing, with no possibility of recovery.

7 Conclusion

In conclusion, our security analysis of the custom-made "dnsProxy" server unveiled critical vulnerabilities. These findings underscore the importance of robust security measures in server development. We explored preventive measures such as rate limiting, the token bucket algorithm, and input validation to mitigate potential risks. Our exploration provides valuable insights into securing server applications, emphasising

the need for vigilant development practices and proactive defences against evolving cybersecurity threats.

References

[1] Ray, D., Ligatti, J.: Defining code-injection attacks. Acm Sigplan Notices **47**(1), 179–190 (2012) https://doi.org/10.1145/2103621.2103678

[2] Lhee, K.-S., Chapin, S.J.: Buffer overflow and format string overflow vulnerabilities. Software: Practice and Experience **33**(5), 423–460 (2003) https://doi.org/10.1002/spe.515

[3] Jaafar, F., Nicolescu, G., Richard, C.: A systematic approach for privilege escalation prevention. In: 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 101–108 (2016). https://doi.org/10.1109/QRS-C.2016.17

[4] Lenaerts-Bergmans, B.: What is privilege escalation? (2022). https://www.crowdstrike.com/cybersecurity-101/privilege-escalation/

[5] Cloudflare: What is a denial-of-service (DoS) attack? (2024). https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/

[6] OWASP: Denial of Service Cheat Sheet (2024). https://cheatsheetseries.owasp.org/cheatsheets/Denial_of_Service_Cheat_Sheet.html

[7] Docker: Run the Docker daemon as a non-root user (Rootless mode) (2024). https://docs.docker.com/engine/security/rootless/

[8] MrEmpy: MBROverwrite (2023). https://github.com/MrEmpy/MBROverwrite