Software Vulnerability Exploration

Justin Wasser

University of Maryland Global Campus

INFA 670: Information Assurance Capstone

Professor Boyles

3/5/2024

Introduction

This paper aims to explore two of the most common software vulnerabilities, and how malicious actors exploit these vulnerabilities in an attack. Moreover, the ensuing analysis will include explaining the vulnerabilities, where they occur, how they are exploited (with an example attack), the potential ramifications of the vulnerabilities being exploited, and lastly how to prevent or mitigate an attack. To that point, buffer overflow vulnerabilities and attacks will be analyzed in addition to SQL injection vulnerabilities and attacks.

Buffer Overflow

To begin this discussion earnestly, we must first define a buffer. To that point, a buffer refers to a section of RAM dedicated to temporarily storing data being processed/transmitted by a program/software (*Buffer*, n.d.). Moreover, a buffer overflow vulnerability occurs when a program allows an input to be accepted/duplicated without checking to see if the given input will overwhelm the RAM dedicated to the buffer and cause it to “overflow” (Mitre, 2006b), into adjacent memory often used to hold program instructions, which can harm a program/system in numerous (Mitre, 2006b). To that point, by overflowing the buffer with their input, the malicious actor seeks to overwrite the program instructions and replace them with their own, which allows them to inject and then execute the malicious code of their choosing (Cobb, 2021). Moreover, a successful buffer overflow attack can affect all aspects of a program’s CIA, i.e. they could crash the program, or they could execute code that reveals information about or alters the program (Mitre, 2006b). Buffer overflow vulnerabilities can occur in C, C++, and assembly programming languages (Mitre, 2006b). The reason that buffer overflow vulnerabilities are limited to these languages is because they do not have “input buffer” (Mitre, 2006b) safeguards built-in, such as preventing the acceptance of data that is larger than the section of memory dedicated for a given buffer (Mitre, 2006b). Lastly, there are different types of buffer overflow vulnerabilities, but we will not be differentiating between them in this paper (Mitre, 2006b; OWASP, n.d.).

An example of an attack that successfully leveraged a buffer overflow vulnerability was the "Morris worm" (HYPR, n.d.). The Morris worm exploited numerous vulnerabilities in UNIX programs to spread a worm across the internet (HYPR, n.d.). One of the exploits involved attacking a buffer overflow vulnerability in the UNIX program that stored the information (including early versions of email addresses) of users on a system, known as the "finger" (HYPR, n.d.) program (HYPR, n.d.; *In Unix, What Is the Finger Command?*, n.d.). The information gathered by buffer overflow attacks on finger programs was used to help determine where to send the worm next so that it could continue to spread (HYPR, n.d.).

To illustrate a buffer overflow vulnerability, and how to remove the vulnerability, we will observe the code found below:

Example 1:

```
"void manipulate_string(char * string){
char buf[24];
strcpy(buf, string);
...
}" (Mitre, 2006b)
```

The vulnerability in this code is the use of the 'strcpy' function which has no limit on the string input into the buffer (Mitre, 2006b). To that point, one solution to this vulnerability is to substitute the 'strcpy' function with a similar function that allows for limitations to be placed on the length of the string that can be accepted into the input buffer, such as "strncpy" (Mitre, 2006b). Correctly applied, this change in function mitigates the buffer overflow vulnerability as any input larger than the memory allocated for the buffer will be disallowed, thereby preventing overflow into neighboring areas of memory (Mitre, 2006b). Moreover, only writing code with restricted functions is proven to minimize the occurrence of buffer overflow vulnerabilities (Mitre, 2006b).

Additionally, another mitigation technique involves utilizing programming languages that “perform their own memory management, such as Java and Perl” (Mitre, 2006b). This approach prevents the occurrence of buffer overflow vulnerabilities by using a programming language that by default does not allow for such vulnerabilities to be coded into existence (Mitre, 2006b). Moreover, another mitigation technique is to only grant the privileges necessary to accomplish the required task (Cobb, 2021). Implementation of this security control would discourage buffer overflow attacks as the harm that could arise from successful attacks would be very minor as user accounts would only have access to their information and therefore attacks would not be worthwhile from an attacker’s perspective (Cobb, 2021). Lastly, a final mitigation technique is “Address space layout randomization” (Cobb, 2021). Utilizing this technique makes it more difficult for attackers to predict where the executable portion of the memory is located, making it more challenging for malicious actors to craft necessary inputs that will harm the software, which minimizes the risk of a buffer overflow attack being successful (Cobb, 2021; Mitre, 2006b).

SQL Injection

For starters, SQL stands for “Structured query language” (Amazon AWS, n.d.) which is a programming language specifically created to optimize communication with relational databases (Amazon AWS, n.d.). Moreover, an SQL injection vulnerability is present in applications (often web-based) that are connected to a SQL database and take potentially harmful, unrestricted user inputs (data) and incorporate them into code that produces SQL commands (kingthorin, n.d.;

Mitre, 2006a; OWASP, 2013). To that point, this architecture allows users to input strings (via a data input field such as a login window), including data that have the syntax to be interpreted as SQL to the underlying relational database and thereby alter the instructions/logic of the SQL code to produce a different SQL request than the database owner/architect intended (kingthorin, n.d.; Mitre, 2006a; OWASP, 2013). Moreover, the presence of this vulnerability allows malicious actors to alter or evade the security parameters (such as authentication controls) to gain unauthorized access to the database which they can leverage in numerous ways that harm the CIA of the information within the database and restrict the ability to for database owners to access the database (kingthorin, n.d.; Mitre, 2006a). To that point, the harm that can arise from a successful SQL injection includes disclosure/theft of confidential data, altering of data, deletion of data, and/or loss of access to the database (kingthorin, n.d.; Mitre, 2006a; OWASP, 2013).

An example of an attack that successfully exploited an SQL vulnerability was the attack on HGGary, which resulted in the disclosure of user authentication data that was leveraged to crack hashed passwords (followed by additional steps), ultimately resulting in the company's servers being breached and "its data destroyed, and its website defaced" (ARS Staff, 2011) (ARS Staff, 2011). Moreover, the SQL injection portion of the attack was made possible by HGGary's website utilizing a vulnerable "content management system (CMS)" (ARS Staff, 2011) that itself utilized a SQL database (ARS Staff, 2011). The CMS's SQL injection vulnerability was exploited by malicious actors altering the URL's "pageNav" (ARS Staff, 2011) and "page" (ARS Staff, 2011) parameters which resulted in the CMS disclosing confidential information stored within its underlying SQL database to the attackers (ARS Staff, 2011).

To illustrate an SQL injection vulnerability observe the following code:

Example 2:

```
“string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName + "' AND itemname =
'" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);” (Mitre, 2006a)
```

The vulnerability in this code is the use of unsecured user data as part of the code used to produce SQL search parameters/commands (Mitre, 2006a). This allows a malicious actor to use the user input window to alter the SQL search parameters by writing **’ OR ‘5’=’5** for example (or any other universally true statement after ‘OR’ with the syntax shown) as the data designated as **itemname** thereby effectively breaking the intended purpose of the code by making the entire condition evaluated as true regardless of the authenticated username (Mitre, 2006a; OWASP, 2013). The result is that all items in the database (regardless of owner) will be disclosed to the attacker (Mitre, 2006a).

There are several techniques for minimizing, mitigating, and/or preventing an SQL injection, one of which is to write the code with a more restrictive architecture that does not allow for user inputs to access SQL command functions, known as “Parameterization” (Mitre, 2006a). To that point, constructing code using parameterization removes the vulnerability of combining user data inputs with SQL commands thereby preventing user data from being interpreted as containing an SQL command even if it contains SQL syntax (Mitre, 2006a; *SQL Injection Prevention Cheat Sheet*, n.d.). For example, given the code illustrated in “Example 2”, if the code were written using parameterization then even if a malicious actor entered **’ OR ‘5’=’5** as their itemname, the code would execute as intended and search for items designated as **’ OR ‘5’=’5** under whichever username the attacker entered, thereby removing the SQL injection vulnerability (Mitre, 2006a; *SQL Injection Prevention Cheat Sheet*, n.d.).

Additionally, another technique for minimizing/mitigating an SQL injection is to limit the privileges given for the task, i.e., only granting the privileges necessary to accomplish the task (Mitre, 2006a). Moreover, implementing this security control would only allow users to access their data, which would mitigate the harm of a successful SQL injection as the malicious actor would only be able to view/alter data associated with the username used (Mitre, 2006a). Lastly, a final SQL injection mitigation technique is to create a list of acceptable user inputs and reject all other inputs that do not exactly match the list of acceptable inputs, which will necessarily stop almost all malicious SQL injection attempts (Mitre, 2006a).

Conclusion

In conclusion, we were tasked with explaining two of the most common software vulnerabilities, and how malicious actors exploit these vulnerabilities. To that point, buffer overflow vulnerabilities and attacks as well as SQL injection vulnerabilities and attacks were investigated. Moreover, analysis of these vulnerabilities included the origin of the vulnerabilities, where they occur, how they are exploited (with an example of a successful attack), the potential consequences of a successful attack, and lastly how to prevent an attack or minimize the effects of a successful attack.

References

Amazon AWS. (n.d.). *What is SQL?* Amazon Web Services, Inc. Retrieved February 27, 2024, from https://aws.amazon.com/what-is/sql/

ARS Staff. (2011, February 15). *Anonymous speaks: the inside story of the HBGary hack*. Ars Technica. https://arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack/

buffer. (n.d.). PCMAG. Retrieved February 25, 2024, from https://www.pcmag.com/encyclopedia/term/buffer

Cobb, M. (2021, July). *buffer overflow*. SearchSecurity. https://www.techtarget.com/searchsecurity/definition/buffer-overflow

HYPR. (n.d.). *Morris Worm*. Www.hypr.com. Retrieved February 29, 2024, from https://www.hypr.com/security-encyclopedia/morris-worm#:~:text=The%20Morris%20worm%2C%20named%20for

In Unix, what is the finger command? (n.d.). Kb.iu.edu. Retrieved February 29, 2024, from https://kb.iu.edu/d/aasp

kingthorin. (n.d.). *SQL Injection*. OWASP. Retrieved February 27, 2024, from https://owasp.org/www-community/attacks/SQL_Injection

Mitre. (2006a, July 19). *CWE-89: Improper Neutralization of Special Elements used in an SQL Command (“SQL Injection”)*. Mitre.org. https://cwe.mitre.org/data/definitions/89.html

Mitre. (2006b, July 19). *CWE-120: Buffer Copy without Checking Size of Input (“Classic Buffer Overflow”)*. Cwe.mitre.org. https://cwe.mitre.org/data/definitions/120.html

OWASP. (n.d.). *WSTG - v4.1 | Testing for Buffer Overflow*. Owasp.org. Retrieved February 25, 2024, from https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/07-Input_Validation_Testing/13-Testing_for_Buffer_Overflow.html

OWASP. (2013). OWASP Top 10 - 2013. In *OWASP*. https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf

SQL Injection Prevention Cheat Sheet. (n.d.). Owasp.org; Owasp. Retrieved February 28, 2024, from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html