Project Report

Team: 6H0571N7H35H3LL

Abhishek Marathe, Aditi Baraskar, Angadsingh Kalra, Avinash Tahakik, Mitali Kulkarni, Utsav Dusad

1. Introduction

The project was aimed at cracking the CTF challenges using tools and automated scripts. In this project, we wrote scripts, analyzer tools and used existing tools to aid us in CTF. Since there are some vulnerabilities and common techniques to exploit these vulnerabilities, tools were written to automate them. These automated tools and knowledge acquired during previous CTF competition **helped us secure the second spot in the final CTF.**

2. Common Vulnerabilities

Attacks:

- Buffer Overflow:
  - Buffer overflow attacks are well-known and can happen when the boundary conditions for arrays or buffers are not checked properly. This can be used by the attacker to change the Instruction Pointer to execute attacker generated code.
  - This type of vulnerability was observed in the CTF challenge for the “rop_keyboard” service.
- Command Injection:
  - This type of attack can be performed when user input is passed to the system or popen library functions without proper sanitization.
  - This vulnerability was present in the “backup-child” service in the CTF. We used this vulnerability to attack other teams’ “backup-child” service and tried to gain access to /bin/sh. Once we got the shell access, we did not leave the shell, so as to persistently attack the service without exception of teams who later applied the patch to that service.
- Dot Dot attack:
  - This type of attack can be launched if relative paths are used in the application and application relies on user input to perform file-related operations.

Prevention:

- Buffer Overflow:
  - Buffer overflows can be prevented by meticulously checking the buffer boundary conditions and making sure that off-the boundary access is not performed.
  - ASLR mode can be enabled to randomize the memory address locations so that attacker cannot determine the location of the buffer in which shellcode is kept.
  - Compiler can enable canary mode so that sequential writes all the way to eip can be determined and application can be terminated if the attacker has changed the Instruction Pointer.
- Command Injection:
  - Input sanitization is must to avoid unauthorized access to resources in the system, if the application is using system or popen functions.
  - Instead of system/popen functions user can use execv variants.
  - The patch for the “backup-child” service was very simple. It just needed commenting the system() function call in the code.
- Dot Dot attack:
  - Input sanitization is needed to avoid user from executing dot dot attack and restrict attacker from accessing unauthorized resources.

- Chroot api can be used to restrict the user from getting into restricted parent directories.

3. Implementation Details

In order to be victorious in CTF, one has to make efficient usage of time.Writing automation scripts and tools are two ways to go forward.

- Code analyzer:
  - **Source code analyzer:**

    Some of the functions in the C language are known to be vulnerable(e.g strcpy, gets, system etc) and can be exploited. We created a tool in Python which will analyze the source code and report the functions that are vulnerable.
  - **Binary code analyzer**:

    Similar to source code analyzer, binary source code analyzer was written in Python the only hiccup was to read the binary file, so we used the readelf utility in the linux to get the symbol information in the file and then reported them.

- Network Analysis Tool :

  Since we did not have root access, running wireshark on the server was not possible to monitor the traffic. To circumvent this constraint, we created a tool which will generate the pcap file using tcpdump command and send this file to our local machine where it will be analyzed using Wireshark. By doing this analysis we were able to determine the attacks that other teams were launching onto our services and based on that designed our defense strategy.

- Sanitization :

  Most of the vulnerabilities in applications can be exploited using input that we give to a program. In order to prevent such attacks input, sanitization becomes a necessity.
  - **Sanitization for binary files:**

    If the binary file is provided, and we do not have access to the source code file, we cannot update the source code by inserting some function that sanitizes the input, recompile it and deploy the application. An alternative way was needed to overcome this challenge. We resolved it by designing a tool with the help of pwntool library's tube functionality, which will run as a middle man in between service and executable binary. This wrapper will take input from user, sanitize that input and then pass it forward to the executable binary. Similarly, the output of the binary was again relayed back to the user. In this way, we can sanitize every user input.
  - **Sanitization for source files:**

    If source file was given for the application, then sanitizing the user input becomes relatively easy as we can write the sanitize function, which will process the input provided by the user, and insert it at every location when user input is received.

- Scripting :

  Writing a script to launch the attacks on multiple teams in parallel is as important as figuring out the vulnerability and then exploiting it in order to win the CTF. Which is why we had decided to write the scripts based on experimental CTF. These scripts helped us during the final CTF to automate our attacks very efficiently.

- Shell Code generator :

- As time is critical in CTF, and buffer overflow attacks are quite frequent. Most of the time these attacks are exploited using shellcode injection, shellcodes differ for different machines as well as commands. We wrote an application which would generate the shellcode for x86 architecture(32 bit). It can generate shell code for system(/bin/sh), execve etc. functions.

- Binary application tools:
  - **Tool to find gadgets in binary:**

    Finding gadgets in a very large binary can be very tedious. We created a tool in python to find specific gadgets in the binary. The tool takes two inputs, the name of the binary and the instruction list for the gadgets that you need find.
    Example: python findRopGadget.py --filename sample_c --instructions mov,pop
    will find all the gadgets which pertaining to the instructions mov and pop in the binary sample_c.

- Tools for Web:

  We created a basic script in python to find potential vulnerabilities in a php file. It looks into the following three vulnerabilities:
  - **SQL Injection:**

    The function findAllVulnerableQueries() in the script finds the vulnerable queries in a given php file. This tool finds all the sql queries using a regular expression and tries to classify it as vulnerable if it finds any user controlled variable in the query or register globals like the request parameters from $_GET or $_POST.
  - **XSS:**

    The function findPotentialXSSVulnerability() in the script finds potential XSS vulnerabilities. This is just restricted to find XSS that can be carried out through the echo function. The above given function checks if any user controlled variable occurs along with the echo function. It does not handle any other type of XSS yet.
  - **OS Command Injection:**

    The function areVulnerableFunctionsPresent() checks if functions like eval and system are present in the php file along with the user controlled variable. The hardcoded list of vulnerable functions is present in the script.

- **Defense for Web attacks:**
  - **SQL Injection:**

    We planned to install the htmlpurifier library to handle sql injection http://repo.or.cz/w/htmlpurifier.git .

    ```
    <?php
    require_once 'HTMLPurifier.auto.php'; $purifier = new
    HTMLPurifier(); $clean_html = $purifier->purify($dirty_html);
    ?>
    ```

    The above script will clean up the invalid nesting, unsafe attributes and tags like <script> on click will be removed.

  - **OS Command Injection:**

    PHP provides some built-in functions to sanitize inputs which can help in protecting against command injection:
    escapeshellarg($str): adds single quotes around a string and quotes/escapes any existing single quotes allowing one to pass a string directly to a shell function and having it be treated as a single safe argument

escapeshellcmd($str): escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands (#&;`|*?~<>^()[]{}$\, \x0A and \xFF. ' and " are escaped only if they are not paired)

- **XSS:**

  Defence: We can change the server configuration such that the remote url inclusion (set allow_url_include to '0'.) is not allowed. This would prevent us from remote XSS attacks.

- **HTML Encoding:**

  PHP provides htmlentities() function converts all the applicable characters to HTML entities. In javascript, we can write the following function to encode the HTML entities using the following function.

```
var encodedStr = input.replace(/[\u00A0-\u9999<>\&]/gim, function(i) {
                                        return '&#'+i.charCodeAt(0)+';';
                                    });
```

4. Tools Usage in actual CTF:

- **Tools used:**
  1. Our code analysis tools really helped us to detect the vulnerable spots in the services, especially for analyzing the “ROP_keyboard” binary in which gets was the vulnerable function.
  2. Scripts written to launch attack on multiple teams once the exploit was discovered were tremendously useful.
  3. Command line injection script, was used to launch automated attacks to exploit vulnerability in backup-child service.
  4. Script written to check for possible vulnerabilities in a php file was used to check the securedb.php file provided in the securedb challenge. The script could indicate that there was no sql injection, OS command injection or xss vulnerability. But our script did not handle the presence of variable variables. Hence, we could not identify the potential vulnerability there.

- **Tools weren’t used**
  1. Some of the ideas we had thought beforehand, such as middle-man application for input sanitization could not be done because root access was not provided.
  2. Shell code generator was not used as buffer overflow exploit was not present in any of the service.
  3. Script written to analyze the network traffic was not used as reading access logs file gave every detail we needed to figure out the attack launched by the other team and relaunch it.

5. CTF:

- **Workload distribution:**
  - Since there were three services, we assigned two teammates to each service based in their expertise in the field to deal with each service.
  - Every member of the team contributed to write the tools which were used during the CTF.
  - Since writing scripts for automating attacks and finding vulnerabilities was an integral part of the CTF challenge, we all had prepared well and each one of us was capable

of automating the attacks by writing or modifying the scripts as necessary for the respective attack.

- **CTF challenges:**
  - No root access:
    - As root access was not provided, packet analysis using Wireshark was not possible. Which precluded us from doing packet analysis on VM.
  - No permission to edit wrapper.sh
    - Since only binary was given for some challenges and was executed using wrapper.sh, input sanitization was not feasible as explained in “Sanitization for binary files”.
    - If root access was provided then we could have modified the wrapper.sh to allow our application in the middle to sanitize the input.
    - Figuring out the system function address to put in eip for ROP_keyboard was a bit challenging and consumed a lot of time.

  - **Improvements:**
    1. Functionality of the static code analyzer could be improved to analyze buffer overflow attacks.
    2. ROP attack requires gadgets in the binary, gadget finder application can be improved to find more sophisticated gadgets and launch more serious attacks.
    3. Shell code generation script can be improved to cater multiple architectures, operating system and functions.
    4. We could enhance the script for finding vulnerability in PHP to also identify the presence of variable variables.
    5. We could enhance the code analyser script to include more functions.

  - **Tools improvisation after the CTF**
    - As service was reading a file whose name is provided as a user input.
    - Normal attack would have been providing <flagid>_* so that we can get flag
    - Instead of giving input on every service run, we used command line injection to get shell. Also, we kept the connection alive after getting a flag, so that even after other team apply a patch, our attack won’t get affected.
    - Web attack: In the pre CTF, we changed PHP in such a way, that the post request from the CTF admin would send 200 OK and would not save the the data sent from the CTF admin. This helped to us defend our system since we do not have the FLAG on our machine then others couldn’t get the flag from our system. In the Final CTF we could have done the similar thing. By checking the logs we can see what the CTF admin is sending. And in the same function we do not allow the CTF admin to save the FLAG.

  - **Lessons learned from the project / CTF experience :**
    - **Importance of security in application**

      CTF and project taught us the importance of the security in the applications and how attackers can exploit these vulnerabilities to do malicious activities.The ways with which one can carry out various attacks to exploit the vulnerabilities in the application.
    - **Importance of Tools:**

      Tools and their usage is an important thing an attacker must know to exploit various vulnerabilities, tools will allow to carry out attacks faster and in much more sophisticated way.

- **Not to give up:**
  - **Attacker has to get successful only once**, so no matter how daunting the challenge is sticking to basics and doing things logically does help to find vulnerabilities in the applications
- **Importance of detailed knowledge about the system:**
  In order to exploit the application, detailed understanding of the underlying system is important.

References:

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
http://php.net/manual/en/index.php
https://stackoverflow.com/questions/18749591/encode-html-entities-in-javascript
https://github.com/dustyfresh/PHP-vulnerability-audit-cheatsheet
http://repo.or.cz/w/htmlpurifier.git

Credits:

We would like to thank Adam for being such a cool teacher, the other Adam and Connor for helping us throughout the semester.