White box testing of software “Santa’s PyHelper”

Bruno Caseiro
Departamento de Eletrónica, Informática e Telecomunicações
Universidade de Aveiro
Aveiro, Portugal
caseiro@ua.pt

***Abstract*—Background: Santa’s PyHelper is an unreleased software with the purpose of handling Christmas gift lists. The program can handle logins and account creations. The white box testing technique was applied to test the security of Santa’s PyHelper software.**
Results: There were four vulnerabilities found, one of each of the following categories: improper input validation, exposure of sensitive information to an unauthorized actor, improper authentication and OS command injection. All of these are in the “2020 CWE Top 25 Most Dangerous Software Weaknesses”[1]. All of them can be patched with simple modifications to the software’s source code.
Conclusions: Santa’s PyHelper is a heavily flawed software that should not be released to the public just yet. The program has failed several simple, basic security tests.
Keywords—vulnerabilities, validation, injection, unauthorized

I. Introduction

The software under testing has been recently coded and never tested before. It was expected for some flaws to be detected. The application works offline but still has a built in account manager mechanism which is heavily flawed. The original goal was to be able to create accounts and login without connecting to the internet. Although these features work, there are many bugs around them which could have been corrected by simple modifications to the code.

After logging in it’s possible to manage a gift list (add, edit, modify. It’s also possible to create new users, log out and change accounts at any time.

II. Problem description

A. Improper input validation

When opening the application, a menu is presented to the user. A couple of numbered options are shown along with the question “How can I help you?” and a decorative Christmas tree. An arrow points to a blank space, which implies the program is waiting for a user input. It’s very intuitive that the user is supposed to type in a number. If the user types an integer that isn’t shown in the menu e.g. “4”, when there are only three options, the program returns an error message and rightfully so, prompts the user to input a valid number.

However, if the user inputs another character that is not an integer, the software crashes and shows an error message, from which an attacker could easily infer the language in which the program is written as well as investigate some of the source code dumped with the error message. This is the result of improper input validation.

```
Traceback (most recent call last):
  File "santa.py", line 125, in <module>
    choice = int(printMenu(False))
ValueError: invalid literal for int() with base 10: 'a'
```

Fig. 1. Entering ‘a’ as input when the main menu requests an integer. Example of improper input validation

There are other cases in which improper input validation could have happened, but this is the only case in which the input isn’t properly sanitized. For that reason, it’s assumed the architecture of the software was properly planned but wasn’t well executed in the implementation phase. This vulnerability is of low to medium severity and has a CVSS base score of 4.0 as it only slightly impacts the availability metric for a short time span.

B. Exposure of sensitive information to an unauthorized actor

In the previously mentioned main menu, option 2 represents the login operation. After selecting it, if a valid, existing username is typed in, the system requests the corresponding password. On the other hand, If a random sequence of letters is typed in as the username (not present in the database), the current user attempting to log in is left with the message ”Username does not exist!”.

This vulnerability allows for user enumeration, facilitating part of the process of an attacker finding users’ credentials. This vulnerability is also low to medium severity and has a CVSS base score of 4.0. It slightly impacts the confidentiality metric.

C. Improper authentication

After choosing option “2. Log in” and entering a valid username, the user is asked for a password. If an incorrect password is given, the password prompt reappears. After an incorrect password is given as input three times, the login is completed successfully. This means that the password wasn’t even required to log in. This is a huge hole in security that had to be discussed during the project inception phase.

Problem C of type “improper authentication” is a major vulnerability, with a CVSS base score of 9.0 as it heavily impacts confidentiality and possibly integrity of user data.

D. OS command injection

As the white box method was used to test the application, it’s known that the following line creates a folder in the software’s directory after registering a user.

```
os.mkdir(os.getcwd()+ "/" + user)
```

Fig. 2. Line 43 of Santa’s PyHelper, representing how a folder is created for a recently registered user.

978-1-5090-6008-5/17/$31.00 ©2017 IEEE

Since the variable "user" is an input from the keyboard, it's fairly easy to inject OS commands during account creation.

This vulnerability is extremely dangerous, having a CVSS score of 9.2 as it possibly highly affects each of the CIA metrics.

III. Problem detection

A. Improper input validation

As seen in Fig. 1., problem A can fail a test when an input other than an integer is passed to the program. Many characters were tested, some of them being 'a', '?', '!', '.', '1', '0'. The first four of this list all returned the same error and resulted in a crash, while the last two successfully completed the test, as expected.

It is also easily detectable by just a quick glance at the code. There's no validation, instead a cast to integer is done directly on the user input, which can easily result in an error.

B. Exposure of sensitive information to an unauthorized actor

The fuzzing technique can be applied to enumerate several, if not all users in a database. An attack run by the previously mentioned technique would probably return satisfying results for an attacker, leaving only half of the credentials to be discovered (the passwords).

A simple test that was done as proof of concept was typing in random characters as a username, which resulted in the message "Username does not exist!".

When attempting to log in with an existing user, only the password was requested.

In the actual code, there's an immediate validation of the username before requesting the password. If the user does not exist, the login process is interrupted with the warning "Username does not exist!".

C. Improper authentication

At first, many techniques were used to attempt to bypass the account authentication phase. One user "test" was created for this purpose, and when logging in and attempting some payloads, after only three attempts there was some speculation that it had resulted in a bypass of the login. However, after repeating the process with meaningless strings as passwords, it was concluded that any input would bypass the authentication as long as the three password tries were used.

It's easy to detect this flaw when analyzing the code as the loop which repeats the form in case of failure revolves around the number of tries instead of the validation of the password. This alone isn't a problem, however after the loop is done (no tries left) the program assumes the login was successfully completed.

D. OS command injection

The last vulnerability is "well hidden" if we don't look at the code, but it was still found when inspecting the files inside the software's directory. Folders are created for each separate user, leading to speculation that an OS command (mkdir) is called using user input. The theory was confirmed after inspecting the code, as seen on Fig. 2.

First, a legitimate user was registered with the username "test2". The program's directory was analyzed and as expected, a folder "test2" was created.

Trying to inject code in the account creation process, a user was registered with the username "../injectTest". There were no new folders in the software's directory, however, in the parent directory the folder "injectTest" had been created. At this point, it was confirmed that a OS command injection vulnerability was present in the system.

IV. Problem Solution

A. Improper input validation

A simple input validation mechanism would be enough to prevent this type of crash. Allowing the user to only input integers and looping the prompt until a valid input was given is a valid solution.

In greater detail, a try/catch block would be ideal, capturing a "ValueError", which is an exception raised when an input of a wrong data type is given.

Current	choice = int(printMenu(false))

Proposed Solution	try:
	  choice = int(printMenu(false))
	except ValueError:
	  print("Please input a number")

Fig. 3. Solution for problem A. Improper input validation

B. Exposure of sensitive information to an unauthorized actor

A fairly easy fix is to let the user input the password even after typing an unexisting username. If that's the case, a message similar to "invalid credentials" would be sent to the user.

With this technique there is no way to identify which of the inputs was wrong or even if the given username exists. It's still possible to enumerate usernames by creating accounts with existing usernames, but it's much harder for an attacker.

Current	while(user not in userList):
	  print("Username does not exist!")
	  user = input("\nUsername: ")

Proposed Solution	user = input("\nUsername: ")
	pwd = getpass("\nPassword: ")
	
	while(invalidCredentials(user, pwd)):
	  print("Invalid credentials!")
	  user = input("\nUsername: ")
	  pwd = getpass("\nPassword: ")
	
	# The invalid credentials functions would verify if the user exists, and if it does, check if the 'pwd' argument matches

Fig. 4. Solution for problem B. Exposure of sensitive information to an unauthorized actor

C. Improper authentication

One of the two biggest vulnerabilities in the system and seen as a forgotten feature during the inception phase of the software development, the improper authentication in Santa's PyHelper is poorly programmed since the loop focuses on the number of tries, instead of verifying if the passwords match.

The following is one possible solution, but not a unique way of solving this vulnerability.

Current	tries = 3
	while(tries != 0):
	  pwdAttempt = getpass("Password: ")
	  if (pwd != pwdAttempt):
	  print("Wrong password\n")
	  else:
	  break
	  tries -= 1
	….
	return user

Proposed Solution	tries = 3
	pwdAttempt = getpass("Password: ")
	
	while(pwd != pwdAttempt):
	  tries -= 1
	  if (tries == 0):
	  print("Authentication failed")
	  return False
	  print("Wrong password\n")
	  pwdAttempt = getpass("Password: ")

Fig. 5. Solution for problem C. Improper authentication

D. OS command injection

Cross-site scripting and OS command injection are similar vulnerabilities which rank first and sixth, respectively, in "CWE's Top 25 Most Dangerous Software Weaknesses"[1], and for good reason. It's very hard to avoid code injections on web applications, but the same cannot be said about OS commands.

The ideal solution would be to not use the 'os' library and avoid calling system commands. If it really is a necessity, it's justifiable that the user is severely restricted on it's input. The following solution only allows alphanumeric characters to be used as input, and therefore prohibiting symbols and other characters.

Current	user = input("\nUsername: ") …. os.mkdir(os.getcwd() + "/" + user)

Proposed Solution	user = input("\nUsername: ")
	while(not user.isalnum()):
	  print("Please only use alphanumeric characters")
	  user = input("\nUsername: ")
	….
	os.mkdir(os.getcwd() + "/" + user)
	
	# str.isalnum() is a Python function which only returns True if str contains only alphanumeric characters False if otherwise

Fig. 6. Solution for problem D. OS command injection

Discussion of results

Although only four vulnerabilities were found in the system, it's very likely more security flaws are present as it is still under some testing and prone to major changes.

The first vulnerability, improper input validation when choosing a menu option isn't seen as a huge problem since it only results in a crash. There's only one affected user in this scenario, the local, offline user. It does not compromise the integrity of other accounts or gift lists and for that reason it's not a priority, but an easily fixable bug which would make the software a little bit more robust.

The second problem can happen not only in the login page, but also in other menus. The same technique can be used in the "1. Create an account" option as obviously, it's not possible to create an account with an already existing username and that message must be shown to the user. It's not a viable option to return a generic message since it won't be clear what the problem was during the registration phase. The registering process is fairly quick and for that reason, fuzzing can also be used to discover some usernames. Many tools can be used to exploit this issue, including OWASP ZAP and/or Burp.

Authenticating with no password is a huge problem and should be fixed as soon as possible. No tools are required to exploit this issue.

OS command injections won't be very useful for an attacker since it's assumed he/she already has physical access to the computer. Santa's PyHelper is an offline application. However it can compromise an entire system through a simple, Christmas gift list application. In future, more advanced versions, this application can eventually be used in stores, malls, etc.. in an isolated machine which locks access to other applications. In this case, this vulnerability suddenly becomes of high severity. Burp Suite can also be used to detect this kind of vulnerability.

It's very likely that during the software development lifecycle there was no threat modeling, very few/none code reviews, and basically no secure coding practices were applied.

Conclusions

The final take after white box testing Santa's PyHelper is that the software is not ready to advance to next phases of development. The implementation plan should be reviewed as well as the architecture of the account managing system. After those two phases are reviewed (and possibly redone), the software should be fully tested again.

References

[1] 2020 CWE Top 25 Most Dangerous Software Weaknesses [Online] Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html