Securecat

Monday, July 31, 2023 3:34 PM

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define COMMAND       "/bin/cat "
#define MAX_SIZE      100

void secure(char *str[])
{
      char *replace;
      unsigned int i;
      char dangerous[] = "#&;`'\"*?<>^()[]{}$,\t\n ";

      /* No absolute path */
      while (*str[0] == '/')
      {
            (*str)++;
      }

      /* No directory traversal */
      while ((replace = strstr(*str, "../")))
      {
            replace[1] = '/';
      }

      /* No shell special chars */
      for (i = 0; i < strlen(dangerous); i++)
      {
            while ((replace = strchr(*str, dangerous[i])))
            {
                  replace[0] = '_';
            }
      }
}

int main(int argc, char **argv)
{
      char mycat[sizeof(COMMAND) + MAX_SIZE + 1] = {0};

      setresuid(geteuid(),geteuid(),geteuid());

      if (argc < 2)
      {
            printf("Usage : %s <file>\n", argv[0]);
            exit(1);
```

```
    }

    if (strlen(argv[1]) > MAX_SIZE)
    {
        printf("Filename too long !\n");
        exit(1);
    }

    secure(&argv[1]);
    sprintf(mycat, "%s%s", COMMAND, argv[1]);
    system(mycat);

    return 0;
}
```

It seems like this code has several vulnerabilities that can be exploited.

1. Directory Traversal
    a. It attempts to replace any instance of "../" with "/". This might be possible to bypass with encoded representation
2. Special Character
    a. It has a list of special characters and replaces them with an underscore. Encoding might be able to bypass this
3. Insecure Command Execution
    a. The program constructs a command using user provided information and then executes the command using system(). I feel like this is the way I should go.

notesreader

Friday, August 11, 2023 4:36 PM

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crypt.h>

#define NOTES          "notes.txt"
#define ENV_AUTH       "AUTH"
#define PASSWORD       "42Oz6uCfSR.SI"

int main(void)
{
      char *auth;
      FILE *notes;
      int c;

      if ((auth = getenv(ENV_AUTH)) == NULL || strcmp(crypt(auth, PASSWORD), PASSWORD) != 0) {
            printf("Restricted access !\n");
      } else if((notes = fopen(NOTES, "r")) == NULL) {
            printf("Fail...\n");
            perror("fopen");
      } else {
            printf("Reading notes :\n");
            while ((c = fgetc(notes)) != EOF) {
                  printf("%c", c);
            }
            fclose(notes);
      }

      return 0;
}
```

This C program is a simple authentication mechanism that reads and displays the contents of a file named "notes.txt". It uses a basic form of password-based authentication.

Here's a breakdown of how the program works:

1. The `#define` directives define some constants:
   - `NOTES`: The filename of the notes file to be read ("notes.txt").
   - `ENV_AUTH`: The environment variable name where the authentication password is stored ("AUTH").
   - `PASSWORD`: The hashed password ("42Oz6uCfSR.SI") that is used for comparison.

2. The `main` function starts executing. It performs the following steps:

   a. It declares some variables:

- `auth`: A pointer to a character array that will store the user-provided password from the environment.
- `notes`: A file pointer that will be used to read the "notes.txt" file.
- `c`: An integer variable used to read characters from the file.

b. It checks if the environment variable `AUTH` is set and not empty, and whether the hash of the provided password matches the stored hash. The password is hashed using the `crypt` function. If the authentication fails, it prints "Restricted access !".

c. If the authentication succeeds, it attempts to open the "notes.txt" file using `fopen`.

d. If the file opening fails, it prints "Fail..." and an error message using `perror`.

e. If the file is successfully opened, it starts reading characters from the file using `fgetc` in a loop. It prints each character to the console as it reads them.

f. After reading the entire file, the program closes the file using `fclose`.

3. Finally, the `main` function returns 0 to indicate successful completion.

In summary, this program is an example of basic password-based authentication that grants access to the contents of the "notes.txt" file only if the correct password is provided through the `AUTH` environment variable. If the authentication fails or if the file cannot be opened, appropriate messages are printed to the console. If authentication succeeds, the program prints the contents of the "notes.txt" file.

Some possible vulnerabilities to look into

- Hardcoded password/Password hashing
- Buffer Overflow
- File Path Manipulation
- File Permission Issues

It looks like I need to get the right authorization code and the right file.

The first step has to be getting the right AUTH variable to gain access
The second step will possibly be overwriting the program so that it doesn't read the notes.txt file but instead reads the flag file.

How I deliver the payload seems to be the issue.

0x8048674 <main+25> call getenv@plt <getenv@plt>
  name: 0x80487e0 ◂— 'AUTH'

Changed ENV_AUTH to ENV_AUTH=1
Won't show up in gdb when looking at the stack.