Lab 1ab Design Project: Attack and Defend Your Shell

Braden Anderson[1] and Alan Kha[2]

[1]bradencanderson@gmail.com, 203744563
[2]akhahaha@gmail.com, 904030522

February 20, 2014

Abstract

Our Lab 1ab implementation is potentially exploitable. This document will show in what ways our implementation is exploitable, and the possible forms that some exploits might take. Lastly, we will discuss security improvements that will protect against malicious attackers.

1 Introduction

Lab 1 implements a shell interpreter for a small subset of POSIX grammar. An input script with valid syntax is read and executed similar to the command `sh script.sh`. There are some key differences between modern shells and our own – most notably, we do not support flow control structures such as loops, if statements, and case switches. As we will discuss in later sections, an absence of loops simplifies design concerns (and, by extension, security concerns) considerably.

Work was shared between team members in this project. For the most part the plan of action was clear, so we alternated days working on the code, occasionally meeting to discuss system design.

2 Security Objectives

The objective of this report is to investigate possible security flaws that a malicious user could use to attack the system running our shell. Vulnerabilities that could be used to harm the host system, in particular using a hostile input script, will be addressed. We will investigate and fix known bugs within our own program, ensuring robust behavior.

3 Overall Design

Our shell interpreter implemenation takes a pipeline approach:

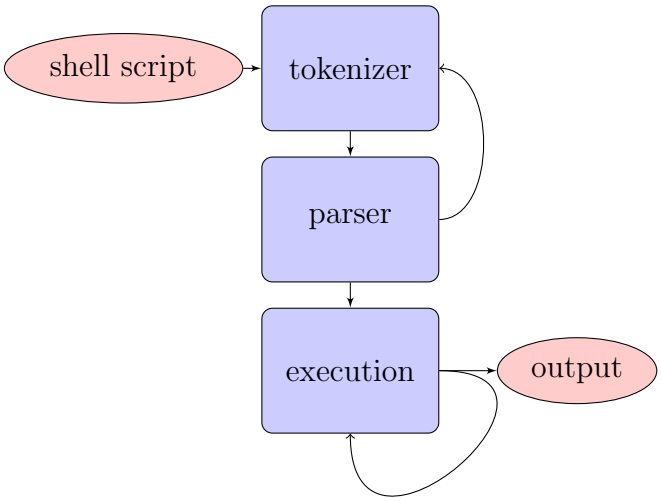
Shell script ⟶ Tokenization ⟶ Parsing ⟶ *Execution*

One exception to the strict pipelining approach is that our parser calls back to the tokenizer when it encounters subshells. For example, tokenizing the following expression:

echo a && (echo b) ⟶ echo | a | && | echo b

In the previous example, `echo b` remains untokenized after the initial tokenization pass. It's stored internally as a "subshell" token, and during the parsing phase it's tokenized dynamically. The parser returns a forest of command trees to main(), and these command trees are executed serially by our execution model. Execution is managed through a recursive traversal of each tree.

So the expanded design for our shell interpreter (without the `-p` option) looks like:



4 Threat Model

This is where we model potential threats. This means that we use our security objectives to transform our overall application design into a system of components, data flows, and trust boundaries. Assumptions that we make:

1. The shell interpreter is trusted, since we wrote all the code. However, it might enter an unsafe state.

2. The user can be stupid or malicious.

3. The operating system is trusted. This means we won't ever be mistakenly given bad memory.

4. The input can be intentionally or unintentionally malicious.

5. Commands that have an interface outside our system – for example, wget – are potential sources of vulnerabilities.

5 Penetration Testing

We performed several penetration tests centered around common vulnerabilities in C and shell scripts.

5.1 Buffer Overflows

Certain C library functions, such as printf(), gets(), and memcpy() have well-known vulnerabilities[1, 2]. This is because the C language does not provide builtin, high-level support for buffers or cstrings. Shell interpreters are also common sources of system instability because they can modify portions of the file system.

Here is a program that is vulnerable to buffer overflows:

```
1  int main(void) {
2          char buf[10];
3          int success = 0;
4
5          printf("Enter string: \n");
6          gets(buf);
7
8          if (success)
9                  return 0;
10
11         return 1;
12 }
```

Most buffer overflows are a result of trying to store data in a buffer of finite size.

During penetration testing we found a bug in our tokenizer where tokens can possibly include nullbytes. We found that a word with an internal nullbyte would be improperly tokenized. However, this behavior does not occur when tokenizing subshells, since we limit our subshell traversal using a call to strlen(), which truncates strings at the first nullbyte. The following table demonstrates this behavior:

input	tokenized representation	output
echo abc	'echo', 'abc'	abc
echo ab\0c	'echo', 'ab\0c'	ab
(echo ab\0c)	'echo', 'ab'	ab

Fortunately we determined that this is not exploitable. This is because in the parsing stage we allocate space for the entire cstring, including nullbytes (as shown in the above

table). So there is no possibility of a buffer overflow attack during the tokenization phase. Here is how we copy and allocate token space, with a dynamically sized buffer:

```
size_t count = 0;
size_t word_size = 8;
char* word = checked_malloc(word_size);

do
{
        // load into word buffer
        word[count] = c;
        count++;

        // expand word buffer if necessary
        if (count == word_size)
        {
                word_size = word_size * 2;
                word = checked_grow_alloc(word, &word_size);
        }
        script++; index++; c = *script;
} while (is_word(c) && index < script_size);
```

All of our code after the tokenization phase deals with nullbyte-terminated strings, so space after the nullbyte is ignored in all strings. Furthermore, there is no vulnerability with the `-p` option because printf() deals with nullbyte-terminated strings.

5.2 Format String Vulnerabilities

We performed another penetration test by looking for format string vulnerabilities. Here are all usages of printf() in our implementation:

```
        printf ("# %d\n", command_number++);
        printf (" \\\n%*s%s\n", indent, "", command_label[c->type]);
        printf ("%*s%s", indent, "", *w);
        printf (" %s", *w);
        printf ("%*s(\n", indent, "");
        printf ("\n%*s)", indent, "");
        printf ("<%s", c->input);
        printf (">%s", c->output);
```

All of the usages of printf() listed here make use of hard-coded format strings, therefore making us unsuccessful in finding a format string vulnerability.

5.3 Inherently Bad Commands

Commands that would harm the system and can be executed through the standard shell can likely be executed through our shell, and are vulnerabilities outside the scope of our security objectives.

A relatively benign yet poignant example of inherently bad commands the ease of deleting the shell interpreter itself, simply by inserting a single line `rm -rf timetrash` into the input script. The script is able to run to completion because the executable has already been loaded

into memory. A slightly more ambitious attack could even delete the entire directory with `rm -rf ../time-travel-shell`. As undesirable as they, these commands are still legal commands that must be obeyed.

5.4 Elevated permissions

Processes in UNIX operating systems inherit the permissions of their parent process. This means that when executing the shell interpreter with superuser permission, great care must be taken to ensure that the input script is safe.

Normally, executing a script with the line `ls > /home/output` would return a failure as creating writing to the home directory requires superuser permission. However, when executing `sudo ./timetrash script`, the redirect commands successfully execute with root access.

5.5 Memory efficiency

The tokenization phase copies the entire shell script into program memory. This means that if a 1GB file is read, we are storing cstrings taking up 1GB of memory. This will result in slow performance and system instability (if the operating system can't allocate more memory), but it is not potentially exploitable. However, the length of the input file is limited to `INT_MAX`, which is system-dependent.

6 Robustness Analysis

- All user input is validated. Illegal characters or incorrect syntax return a parsing error.
- No fixed-size buffers are used in our implementation. Buffers are dynamically resized as more space is needed. The only problem with this implementation is that it can potentially
- There are no calls to printf() with an uncontrolled format string.
- Unsafe C library functions such as strcpy() are not used.

7 Conclusion

We have determined that it is unlikely for an attacker to be able to exploit our processing of the input script to create unwanted behavior, at least using buffer overflow or printf vulnerabilities.

The most likely avenue of attack would likely be through inherently bad commands, but preventing these is out of the scope of our program as they are problems inherent in any shell. Administrators who choose to use our interpreter for their system must take care to ensure only trusted users can execute scripts, and take special care when executing with superuser privileges.

References

[1] scut/team teso, Exploiting Format String Vulnerabilities. Stanford University, Version 1.2, 2001.

[2] Aleph One, Smashing the Stack for Fun and Profit. Phrack, Volume 7, 1996.