SIGPwny ECE 297 Project

Project By	Overseen By	Submitted To
Aditya Nebhrajani	Thomas Quig	Professor Miller
avn5		

Fall 2021

https://github.com/nebhrajani-a/taCTF

1 Abstract

This report describes an "automatic" tool for implementing a side-channel timing-style attack on compiled binaries. Timing attacks are often difficult to design around, especially since it may cause poorer performance. Timing attacks are among the most successful forms of side-channel attack, with Meltdown[1] and Spectre[2] as recent examples, and initial implementations of Unix `login` as classical examples. The tool described by this report specifically focuses on a simpler kind of side channel: it uses Valgrind to count the number of instructions executed by a string comparison function for various inputs, and uses the result to predict the string.

2 Background

2.1 Problem Statement

String comparison is a basic operation in many cryptographic applications. Consider a C function written the following way[3]:

```
1  bool insecureStringCompare(const void *a, const void *b, size_t length) {
2   const char *ca = a, *cb = b;
3   for (size_t i = 0; i < length; i++)
4     if (ca[i] != cb[i])
5       return false;
6   return true;
7  }
```

When input strings `a` and `b` are identical, all is well. However, if the strings differ at any character, the function instantly returns, breaking the `for` loop. This makes sense for efficiency, since we're not wasting time checking the equivalence of two strings that are obviously unequal. However, this gives us a way to "brute-force" one of the strings if the other is held constant: we can simply try every ASCII character, and if the `for` loop isn't broken, the current character must be the correct one.

Of course, the natural question is, how can we tell from the execution of a binary file whether the `for` loop has been terminated or not? This is done by counting the number of assembly instructions executed: more instructions would imply that the function did not terminate: so we must have the right character.

Preventing such an attack requires that the function check both strings all the way through, regardless of whether they stop matching at a certain character. A patched function is thus:

```
bool constantTimeStringCompare(const void *a, const void *b, size_t length) {
  const char *ca = a, *cb = b;
  bool result = true;
  for (size_t i = 0; i < length; i++)
    result &= ca[i] == cb[i];
  return result;
}
```

This entirely removes our side channel. Another way that timing attacks are prevented is by making the program sleep for small amounts of time, or perform some other operations. This throws off any tools like ours.

2.2 Current Tools

Given that such a vulnerability is popular among CTF-style reverse engineering challenges, a tool similar in function to the one we detail does exist: PinCTF[4]. However, it uses Intel's PIN to count instructions rather than Valgrind. Intel PIN is notoriously difficult to install and get working, and has a habit of littering any directory it's called in with `inscount.out` files that need to be removed. The author of this paper could only install PIN on a Garuda Linux virtual machine, but PinCTF still did not run correctly; which is much more PIN's fault than the tool's. PinCTF's open GitHub issues are also a result of PIN's weird behavior. Moreover, our tool's functionality is a superset of PinCTF's, with automatic length computation and more options.

In essence, there's a need for a better instruction counting tool that's plug and play. Valgrind is an excellent instrumentation framework, and its tool `callgrind` has an instruction counting function built-in.[1] From testing with multiple binaries, it seems that this is equally as useful for our purposes as Intel's PIN. While the absolute value of instructions counted differs between Valgrind and PIN, the deltas, which are what we care about, agree.

Our tool is called "taCTF" (pronounced either "ta-CTF" or with each letter said out loud). The language chosen is Python. Haskell was considered[2], but ultimately dropped due to considerations of ease of contribution.

The goal while writing this tool was that there should be no `requirements.txt` or *any* environment setup required beyond installing Python 3 and Valgrind (which can be done in a single command on any good Linux distributions.[3]), and running the script. We also try to provide some options that make life easier when doing attacks of this sort, such as reverse-order attacking, known string prefixes, etc.

3 Methodology

Our job is to run Valgrind's `callgrind` on the binary with various inputs, parse the output for the instruction count, then use the instruction count data to reconstruct the string.

[1] The tool `exp-bbv` also has an instruction counting function built in, but it is "experimental" and so we choose to stick with `callgrind`.

[2] Obligatory "A monad is just is a monoid in the category of endofunctors" quote.

[3] Looking at you, Gentoo.

3.1 Parsing `callgrind` output

First, let's look at how `callgrind` counts instructions:

```
$ valgrind --tool=callgrind ./a.out
==43837== Callgrind, a call-graph generating cache profiler
==43837== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==43837== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==43837== Command: ./a.out
==43837==
==43837== For interactive control, run 'callgrind_control -h'.
==43837==
==43837== Events    : Ir
==43837== Collected : 161590
==43837==
==43837== I   refs:      161,590
```

`Collected : 161590` is the information we need. We can grab it with a simple regular expression. Also, we'll use Python's `subprocess` module (built-in) to call Valgrind:

```
def get_instruction_count(test_str, binary_filename):
    with tempfile.NamedTemporaryFile() as tmp:
        command = f'echo "{test_str}" | valgrind --tool=callgrind
        ↪  --callgrind-out-file={tmp.name}\
        {binary_filename}'
        try:
            with subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,
                                  stderr=subprocess.PIPE) as valout:
                valout = valout.stderr.read()
                valout = int(re.findall('Collected : \d+',
                                        valout.decode())[0][12:])
                return valout
        except IndexError:
            pass
```

Note that we could use a `stdin PIPE` instead of `echo`-ing the string then piping it via the shell, however, this approach has known bugs in `subprocess`[4]. We pass `IndexError`'s in case the instruction count isn't found in the output, and the function returns `None`.

What's with the temporary file? `callgrind` writes output files, which we don't care about (we care about the output on `STDERR`), so we just ask `callgrind` to write it to a temporary file the OS will deal with. `tempfile` is also built into Python.

3.2 Finding the Length

Assuming the binary instantly quits if the length of the input string doesn't match that of the secret string, we can find the length by simply trying every length between 1 and some upper limit. By default, taCTF sets this upper limit to 35, but it can be changed with a command line argument. Consider:

[4]The documentation recommends using `subprocess.communicate()` instead, but this author ran out of time and couldn't play with broken Python modules. He did add it to the TODO section of the README though, so perhaps it'll get done someday when he needs to procrastinate.

```
def find_length(binary_filename, maxlen, verbose):
    maximum = -1
    init_counts = get_instruction_count("a", binary_filename)
    length = -1
    for i in range(1, maxlen+1):
        val = get_instruction_count("a"*i, binary_filename)
        if verbose:
            print("a"*i, ":", val)
        if val is not None:
            if val > maximum:
                maximum = val
                length = i
    if init_counts != maximum:
        return length
```

If all the lengths happen to be the same, we return `None` and move on. Not all binaries have this weakness.

3.3 Finding a Single Character

Given a string and an index, we write a function to get the instruction count of the string formed by trying character in a given set at that index. This is trivial:

```
def find_char_at(test_str, location, binary_filename, verbose, charset_code):
    if verbose:
        print("Testing: ", make_bold(test_str, location), "at", location)
    charset = get_charset(charset_code)
    maximum = 0
    bestchoice = ""
    for char in charset:
        test = test_str[:location] + char + test_str[location+1:]
        val = get_instruction_count(test, binary_filename)
        if val is not None:
            if val > maximum:
                maximum = val
                bestchoice = test
        if verbose:
            print("    ", char, ":", val)
    return bestchoice
```

Naturally, there are probably cooler ways to choose `bestchoice`. In the future, the author may add some basic standard-deviation style statistics to make this choice. For now, a `for` loop will suffice.

3.4 Solve

We can finally write a `find-string` function that solves our problem:

```
def find_string(binary_filename, maxlen=35,
                verbose=False, reverse=False,
                lengthgiven=False, length=0,
                charset_code=0,
```

```
                flag_format=""):
    if not lengthgiven:
        length = find_length(binary_filename, maxlen, verbose)
        if length is None:
            print("[taCTF] I couldn't guess the length, sorry. Try -l LENGTH.")
            sys.exit()
    print("Length guess:", length)
    length_diff = length - len(flag_format)
    candidate = flag_format + 'a'*length_diff
    if not reverse:
        for i in range(len(flag_format), length):
            candidate = find_char_at(candidate, i, binary_filename, verbose,
                                     charset_code)
            print(candidate)
    else:
        for i in range(length-1, len(flag_format)-1, -1):
            candidate = find_char_at(candidate, i, binary_filename, verbose,
                                     charset_code)
            print(candidate)
    return candidate
```

This function has all the features taCTF has. It simply checks whether to find the length or if it's been supplied, then calls `find_char_at` in either left-to-right or right-to-left order as many times as required.

3.5 Arguments

This is the easiest part of the code, and is left as an exercise to the reader.[5] We instead detail here what our arguments do and how to use them:

Argument	Long Form	Function	Default
`-h`	`--help`	Help!	
`-v`	`--verbose`	Verbose output, with each iteration printed	False
`-r`	`--reverse`	Try string backwards.	False
`-f`	`--flag-format`	Known flag format to try.	
`-l`	`--length`	Length if known.	
`-c`	`--charset`	Which ASCII character 'set' to try.	0
`-ml`	`--max-length`	Maximum length to check till.	35

We also detail our character codes:

Code	Meaning
0	Lowercase, uppercase, digits
1	Lowercase
2	Uppercase
3	Lowercase, uppercase
4	Lowercase, digits

Punctuation is included by default in all sets.

[5] I've always wanted to do that. If you aren't in an exercise-y mood, feel free to look at the code on GitHub.

4 Results

Our goal of making the tool as painless to use as possible[6] seems to have succeeded. Let's write a weak C program that has the vulnerability:

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define MAX_INPUT_LEN 100

bool insecure_string_compare(const void *a, const void *b, size_t length) {
  const char *ca = a, *cb = b;
  for (size_t i = 0; i < length; i++)
    if (ca[i] != cb[i])
      return false;
  return true;
}

int main(void)
{
  char flag[] = "sigpwny{flag}";
  char input[MAX_INPUT_LEN];
  fgets(input, MAX_INPUT_LEN, stdin);

  if ((strlen(input) - 1) != strlen(flag)) {
    return 0;
  }
  else {
    insecure_string_compare(flag, input, strlen(flag));
  }

  return 0;
}
```

On a sandbox (Arch) Linux machine, all we need to do is:

```
$ sudo pacman -S python3 valgrind
$ git clone https://github.com/nebhrajani-a/taCTF
$ cd taCTF
$ gcc test/example.c
$ ./tactf.py ./a.out
```

The output is:

```
Length guess: 13
saaaaaaaaaaaa
siaaaaaaaaaaa
sigaaaaaaaaaa
```

[6] Finally, we're rid of PIN!

```
sigpaaaaaaaaa
sigpwaaaaaaaa
sigpwnaaaaaaa
sigpwnyaaaaaa
sigpwny{aaaaa
sigpwny{faaaa
sigpwny{flaaa
sigpwny{flaaa
sigpwny{flaga
sigpwny{flag}
```

Maybe with some more options:

```
$ ./tactf.py ./a.out --flag-format "sigpwny{" -l 13 -c 1
Length guess: 13
sigpwny{faaaa
sigpwny{flaaa
sigpwny{flaaa
sigpwny{flaga
sigpwny{flag}
```

Let's also try some other binary that checks the string in reverse order:

```
$ ./tactf.py test/ELF-NoSoftwareBreakpoints -r -l 25
Length guess: 25
aaaaaaaaaaaaaaaaaaaaaaaaS
aaaaaaaaaaaaaaaaaaaaaaaakS
aaaaaaaaaaaaaaaaaaaaaaackS
aaaaaaaaaaaaaaaaaaaaa0ckS
aaaaaaaaaaaaaaaaaaaar0ckS
aaaaaaaaaaaaaaaaaa_r0ckS
aaaaaaaaaaaaaaaaaaT_r0ckS
aaaaaaaaaaaaaaaaaNT_r0ckS
aaaaaaaaaaaaaaaaiNT_r0ckS
aaaaaaaaaaaaaaaaoiNT_r0ckS
aaaaaaaaaaaaaaaPoiNT_r0ckS
aaaaaaaaaaaaaakPoiNT_r0ckS
aaaaaaaaaaaaaakPoiNT_r0ckS
aaaaaaaaaaa3akPoiNT_r0ckS
aaaaaaaaaar3akPoiNT_r0ckS
aaaaaaaaaBr3akPoiNT_r0ckS
aaaaaaaa_Br3akPoiNT_r0ckS
aaaaaaae_Br3akPoiNT_r0ckS
aaaaaare_Br3akPoiNT_r0ckS
aaaaa@re_Br3akPoiNT_r0ckS
aaaaW@re_Br3akPoiNT_r0ckS
aaadW@re_Br3akPoiNT_r0ckS
aardW@re_Br3akPoiNT_r0ckS
aardW@re_Br3akPoiNT_r0ckS
HardW@re_Br3akPoiNT_r0ckS
```

5 Conclusion

We've achieved our goal: writing a clean and easy to use instruction counting tool. There are more features I'd like to add, and at the top of the list is threading. This, however, is a rather complex endeavor with diminishing returns, given how Python's global interpreter lock works. There is perhaps a solution using asynchronous I/O that I'm currently exploring, and may add. Other things to add include:

- Better statistics for deciding which letter to choose.
- Treat program's movement as a tree and allow going back up the tree and branching in case instruction counts are all equal after a certain choice.
- Add PIN support. Yes, PIN is difficult to use, but having support for PIN gives us another tool to confirm Valgrind's results with.
- Use `subprocess.communicate` instead of `echo` to talk to Valgrind. This will allow punctuation characters that are currently blocked due to `echo`'s limitations and escape character pain.

I had a lot of fun writing this small and simple tool, and hopefully you had as much going through this report. Any suggestions, issues, comments, are welcome on the GitHub issues tracker of this project.[7]

[7] AKA, please, tear apart my code.

References

1. Lipp, M. *et al. Meltdown: Reading Kernel Memory from User Space* in *27th USENIX Security Symposium (USENIX Security 18)* (2018).
2. Kocher, P. *et al. Spectre Attacks: Exploiting Speculative Execution* in *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).
3. Wikipedia. *Timing Attack* `https://en.wikipedia.org/wiki/Timing_attack#Algorithm`. (accessed: 12.08.2021).
4. `https://github.com/ChrisTheCoolHut/PinCTF`.