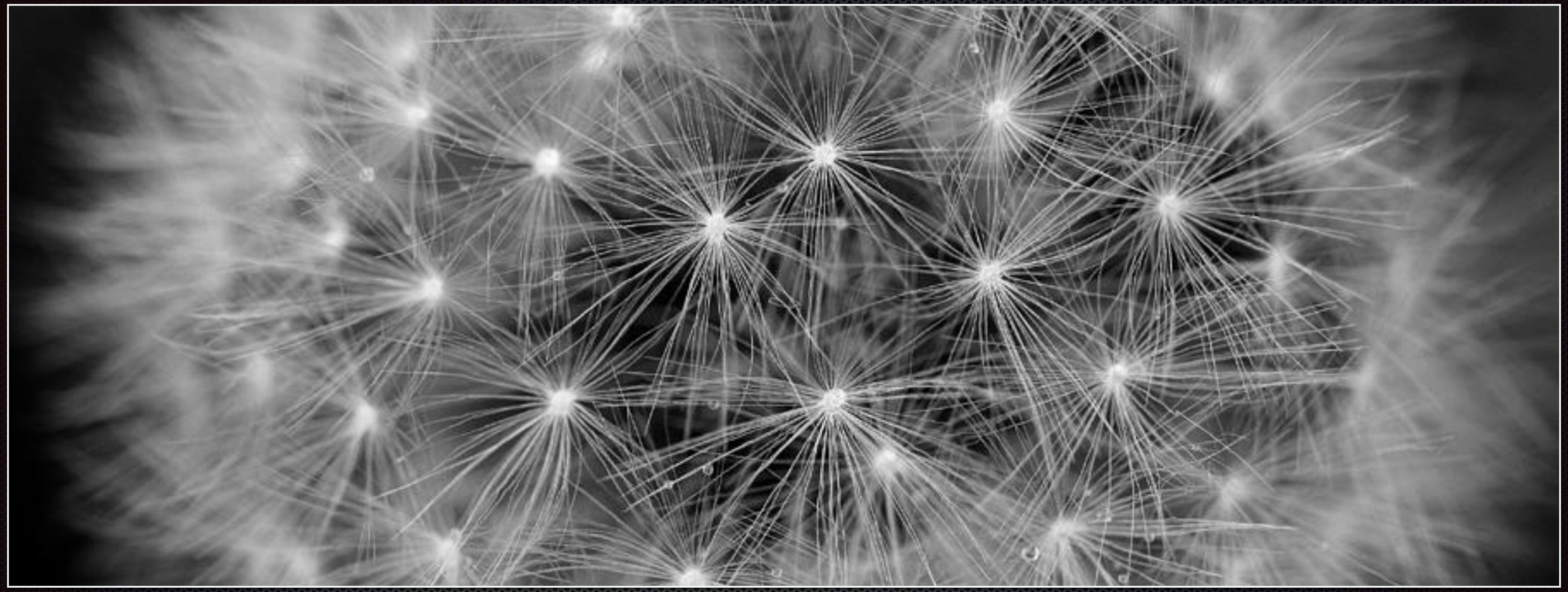

Fuzzing for Vulnerabilities

Mike Ham

Part 2 of Software Exploitation

- In Part 1, you were introduced to processors/RAM/software
  - We did some manual exploit hunting with a little direction
  - Exploited a stack-based buffer overflow vulnerability
- Today we find vulns. more efficiently
- Learn about some new ways to crash programs (exploits up to you)
- Automated tools

How do we find vulnerabilities?

- So many different ways, depends on the nature of what you're trying to exploit
  - Humans, web, software, firewalls
- Source code auditing - you have the original code
- Black box testing - you have the compiled software (more common)

Vulnerability Hunting

- A significant amount of exploitation stems from someone finding a crash in a program and it develops from there
- Some intervention that we have with the program causes a problem
  - Whether or not that problem is something we can use to our advantage is questionable
- Nonetheless, finding a crash is a good place to start in software, we can evaluate if we can exploit that crash later on

Fuzzing

- Sending intentionally invalid data to a product with the intention/hope of finding a crash, error condition, or fault
- Really aren’t any rules, your success in fuzzing is measured purely by the results
  - Banging on a keyboard could be considered fuzzing :)
- The better you can predict what may introduce an error, the better
  - If a prompt asks you for a name, think it’ll also take numbers?

Phases of Fuzzing

1. Identify Target - http://www.securityfocus.com/vulnerabilities
2. Identify Inputs
3. Generate Fuzzed Data
4. Execute Fuzzed Data
5. Monitor for Exceptions
6. Determine Exploitability

Fuzzing stack_overflow_16.c

- Let’s start somewhere simple
- We’ve exploited this program with some direction and were successful
- What if we didn’t have the source code, all we had was just a prompt asking us for some input?
- The problem gets much more difficult since we have no idea how big a buffer is, or what type of data is acceptable for it to take

Fuzzing Strategy

- We could have brute forced it (A, B, C, […], AA, AB, AC) which would take a while but eventually work
- We started to use AAAABBBBCCCC which had letters in 4 byte sequences
  - The idea here was to see if we could cleanly identify that we overwrote some location in memory on a 32-bit machine
  - Works out pretty well, but a unique string would be even better!
- /usr/share/metasploit-framework/tools/exploit# ruby pattern_create.rb 300
- /usr/share/metasploit-framework/tools/exploit# ruby pattern_offset <address>

```
(gdb) run
Starting program: /root/Desktop/basic_exploitation/stack_overflow_16.out
Enter some text:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad
4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8A
g9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad
4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8A
g9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9

Program received signal SIGSEGV, Segmentation fault.
0x62413961 in ?? ()
```

root@kali: /usr/share/metasploit-framework/tools/exploit

File Edit View Search Terminal Help

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ruby pattern_offset.rb 62413961
[*] Exact match at offset 28
root@kali:/usr/share/metasploit-framework/tools/exploit#
```

We can do better…

- The Ruby script in Metasploit is great, it works
- It’s been around for quite some time and it’s pretty slow
  - The script generates a huge unique string every time you run it
  - The string is always the same, so why keep regenerating it?
- We’re going to write a couple more scripts in Python for you to use

stack_overflow_X.out

Can you figure this one out
with what we just learned?

It’s not always too much data…

- Sometimes the *kind* of data can really cause problems for software
- If a program is expecting an integer and you feed it a character, it may crash
  - When programs crash, they sometimes enter into exception handling to help identify errors
  - Exception handlers are vulnerable occasionally too!

Environment Variables

- Both Windows and Linux have them
- Global variables set at the OS level so other programs can read them
- Typically used (set/unset) from the command-line
- Windows: HOME, PATH, USER
  - set
- Linux: printenv

Environment Variables

- Can be changed and set by the user
- If the program is expecting some value, but we can change that value…
- Fuzzing for them and finding out what ones are used is super easy
- Two methods:
  - Using a debugger
  - Automated with script and LD_PRELOAD

Sample Program - Environment Vars

- Run it with GDB and set a break on getenv

```c
// gcc -g environment_variable.c -o environment_variable.out

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char buffer[20]; // create a 20 byte buffer to hold the contents of HOME
    strcpy(buffer, getenv("HOME")); // get the HOME environment variable
    printf("HOME = %s\n", buffer); // print out the value
}
```

```
(gdb) break getenv
Breakpoint 1 at 0x400450
(gdb) r
Starting program: /root/Desktop/basic_exploitation/environment_variable.out

Breakpoint 1, __GI_getenv (name=0x400664 "HOME") at getenv.c:35
35      getenv.c: No such file or directory.
(gdb)
```

Creating Environment Variables

- GENCYBER=AAAA
- export GENCBYER
- If you do not use the export command, the variable will remain local to the shell and won’t work quite right
- Remember, to show them, use: printenv

CTF Link

- http://goo.gl/forms/mivhCv3UELEgmDTM2