FUZZING – PART 1

Mitchell Adair
utdcsg.org
#utdcsg – irc.oftc.net
9/5/2012
Before we begin

• Almost all the material is coming from this book.

• Awesome book, by some of the godfathers of fuzzing

• Buy it…
  
  http://www.amazon.com/gp/product/0321446119
What is fuzzing?

• The process of sending specific data to an application, in hope to elicit certain responses.

• specific?
  • Mutated data, edge cases, etc.

• certain?
  • Crashes, errors, etc.

• Wikipedia
  • “Fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems.”
Why?

- **Used often in**
  - Software testing (SDL)
    - Google, Mozilla, Microsoft and others are huge on this
  - **Bug hunting ← our focus**
    - Good and bad guys, contractors, etc.
    - Looking for fame, profit, etc.
Finding vulnerabilities

• Three primary methods
  • Code auditing
    • Requires source code 😞
  • Reversing
    • No need for source code, focus on binary applications
    • Very time consuming, higher skill degree required
  • Fuzzing
    • Lots of frameworks available
    • Can quickly spin up a custom one yourself (Part 2)
    • Binary, source, etc. unimportant
    • Can be as simple or complex as you like

*After getting used to fuzzing, the problem isn’t finding bugs, it’s figuring out which bugs are exploitable.*
2 days later...

1 libqt4_plugin.dll:0xxxxxxxxx mov ecx,[edx+0x30]
1 msvcrt.dll:0xxxxxxxxx rep stosd
1 ntdll.dll:0xxxxxxxxx mov ecx,[ecx+0x4]
2 libqt4_plugin.dll:0xxxxxxxxx arpl [ecx+0x63],sp
2 libqt4_plugin.dll:0xxxxxxxxx mov eax,[eax+0x18]
2 libqt4_plugin.dll:0xxxxxxxxx or byte [eax+0x10],0x1
3 libqt4_plugin.dll:0xxxxxxxxx mov [eax+0x8],ebx
4 libpacketizer_dirac_plugin.dll:0xxxxxxxxx mov dword [eax+0x20],0x0
4 libpacketizer_dirac_plugin.dll:0xxxxxxxxx mov eax,[edx+0x4]
5 libqt4_plugin.dll:0xxxxxxxxx mov dword [eax],0x1
10 libqt4_plugin.dll:0xxxxxxxxx mov [ebx+0x8],eax
13 libpacketizer_dirac_plugin.dll:0xxxxxxxxx mov edx,[eax+0x8]
2 days later… cont

- 32 unique crashes?
- The ones labeled INVALID, where EIP was invalid look particularly interesting
- 0x00000001 and 0x00000040 especially 😊
Fuzzing phases

1. Identity Target
2. Identity Inputs
3. Generate Fuzzed Data
4. Execute Fuzzed Data
5. Monitor for Exceptions
6. Determine Exploitability

• We’re going to focus on steps 3 through 5
Types of values

- So.. what type of data do I want to send to an application?
- Integer values
  - Border cases:
    - 0, 0xffffffff (2^32)
    - Leverage +n / -n cases
      - malloc( ... + 1)
  - Ranges:
    - MAX32 – 16 <= MAX32 <= MAX32 + 16
    - MAX32 / 2 – 16 <= MAX32 / 2 <= MAX32 / 2 + 16
    - MAX32 / 3 – 16 <= MAX32 / 3 <= MAX32 / 3 + 16
    - MAX32 / 4 – 16 <= MAX32 / 4 <= MAX32 / 4 + 16
    - MAX16 – 16 <= MAX16 <= MAX16 + 16
    - MAX16 / 2 – 16 <= MAX16 / 2 <= MAX16 / 2 + 16
    - MAX16 / 3 – 16 <= MAX16 / 3 <= MAX16 / 3 + 16
    - MAX16 / 4 – 16 <= MAX16 / 4 <= MAX16 / 4 + 16
    - MAX8 – 16 <= MAX8 <= MAX8 + 16
    - MAX8 / 2 – 16 <= MAX8 / 2 <= MAX8 / 2 + 16
    - MAX8 / 3 – 16 <= MAX8 / 3 <= MAX8 / 3 + 16
    - MAX8 / 4 – 16 <= MAX8 / 4 <= MAX8 / 4 + 16

Try to influence sign / unsigned values: char, short, int, etc.

Unsigned value:
\[2^Y\]
Signed value:
\[2^Y / 2\]
Types of values… cont

• String repetitions
  • A*100, A*1000, A*10000
  • Not just A, B makes a difference on the heap, and in hard coded checks!

• Delimiters
  • !@#$%^&*()-_=+{ }\';:"",.<>/?~`
  • Varying length strings separated by delimiters
  • Increasing the length of delimiters is important as well
    • User:..........................................................password

• Format strings
  • %s and %n have greatest chance to trigger a fault
    • %s dereferences a stack value
    • %n writes to a pointer (another dereference)
  • Should include long sequences
Types of values… cont

• Character translations
  • 0xfe and 0xff are expanded into 4 characters under utf16

• Directory traversal
  • If targeting web apps or network daemons
  • ../.. and ..\.. etc.

• Command injection
  • If targeting web apps, cgi scripts, network daemons
  • &&, ; and | characters
Types of targets

- Environment variables
- Positional arguments, flags, etc.
- File formats
- Network protocols
- Web apps
- Etc.
Targeting environment variables

• Focus on *nix because of concept of super user / privilege escalation

• Finding targets
  • We want setuid or setguid files
    • Find / -type f –perm -4000 –o –perm -2000
    • setuid is column 4, value 4
    • setguid is column 4, value 2
    • So setuid and getuid to a normal 640 file would become 6640
Enumerating environment variables

- From GDB, we can hook getenv and show all requests
  - `gdb –q ./myprogram`
    - `break getenv`
      - (say yes if it asks about setting a deferred breakpoint)
    - commands
      - These will get executed after the breakpoint is hit
    - `silent`
    - `x/s *(char **)(ebp+8)`
      - Treat ebp+8 as a pointer to a character pointer
      - Dereference the above, leaves a character pointer
      - `x/s = examine the string of, a character pointer`
      - `Print the string of ebp+8`
  - end
  - cont
Enumerating environment variables...

```
nomnom@ubuntu:~$ gdb -q /bin/mount
Reading symbols from /bin/mount...(no debugging symbols found)...done.
(gdb) break getenv
Function "getenv" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (getenv) pending.
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>x/s *(char **)(%ebp+8)
>cont
>end
(gdb) run
Starting program: /bin/mount
0xf6768a:   "LOCPATH"
0xf67692:   "LC_ALL"
0xf53d57:   "LC_IDENTIFICATION"
0xf67699:   "LANG"
0xf676d0:   "GCONV_PATH"
0xf67692:   "LC_ALL"
0xf53d48:   "LC_MEASUREMENT"
0xf67699:   "LANG"
0xf67692:   "LC_ALL"
0xf53d3b:   "LC_TELPHONE"
0xf67699:   "LANG"
0xf67692:   "LC_ALL"
0xf53d30:   "LC_ADDRESS"
0xf67699:   "LANG"
0xf67692:   "LC_ALL"
0xf53d28:   "LC_NAME"
```
Preloading

• **LD_PRELOAD**
  • “The dynamic linker can be influenced into modifying its behavior during either the program's execution or the program's linking… A typical modification of this behavior is the use of the LD_LIBRARY_PATH and LD_PRELOAD environment variables. These variables adjust the runtime linking process by searching for shared libraries at alternate locations and by forcibly loading and linking libraries that would otherwise not be, respectively.” - Wikipedia
• Creating a substitute for getenv

```c
#define BUFFSIZE 20000
char *getenv(char *variable)
{
    char buff[BUFFSIZE];
    memset(buff, ’A’, BUFFSIZE);
    buff[BUFFSIZE-1] = 0x0;
    return buff;
}
```

• Compiling and running

```
$ gcc -shared -fPIC -o my_getenv.so my_getenv.c
  • Create the shared library, my_getenv.so

$ LD_PRELOAD=./my_getenv.so /usr/bin/target
  • Set the LD_PRELOAD environment varaible and run the application
```
Preloading…

- When the application makes a call to `getenv`, our shared library replacement to `getenv` will get called instead.
Detecting problems

• The poor man’s way
  • Check return code of the application
  • If application terminated due to unhandled signal, return 128 + signal #
    • SIGSEV = 11
    • SIGILL = 4
    • Return values of 132 and 139 should be flagged as possible crashes

• In the previous slide, “echo $?” echoes the return value
  • 139 would be SIGSEV
Detecting problems...

• Better way
  • Use wait or waitpid function
  • fork with an execve in the child and wait or waitpid in the parent
  • can check if child crashed by return value of wait(pid)
Detecting problems...

- The poor man’s (better) way

```python
def monitor(process, logfile, timeout=5):
    # codes of interest
    target_retcodes = {-4:'SIGILL', -6:'SIGABRT', -7:'SIGBUS', -11:'SIGSEGV'}

    starttime = time()
    while process.poll() == None:
        if time() - starttime >= timeout:
            print 'timeout',
            process.kill()
            break
        else:
            sleep(.1) # don't kill cpu %
            continue

    retcode = process.returncode

    if retcode != 0:
        if retcode in target_retcodes.keys():
            logfile.write('\t[!] Program returned retcode (%d, %s)\n' % 
                        (retcode, target_retcodes[retcode]))
            return True  # flag this

    return False  # ignore this
```
Detecting problems...

- Best
  - Use debugging libraries
    - PyDBG, winappdbg, ptrace, etc.
  - Can catch any signals of interest during execution

```python
def Run(program, args):
    # setup pydbg
    dbg = pydbg()
    dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handle_signal)
    dbg.set_callback(EXCEPTION_ILLEGAL_INSTRUCTION, handle_signal)
    dbg.set_callback(EXCEPTION_STACK_OVERFLOW, handle_signal)

    # load program, start it
    dbg.load(program, args)
    dbg.run()

    ...
```

- handle_signal could log important information
  - Register contents, stack values, backtrace, etc.
Targeting file formats (or parsers)

- About half of the local exploits on exploit-db.com are file related
- Common targets include media players, document readers, etc.
Targeting file formats (or parsers)

• What exactly are we trying to exploit with file fuzzing?
  • Format assumptions
    • Length field ABC is supposed to be in the range 0-100, surely no one
      would put 999999…
    • Surely no one would put a negative value…
    • etc.

• Signed bugs
  • Length field XYZ is 4 bytes, and represents the length of the following
    field
    • ie : [….|4 byte size| data of 4 byte size|….]

```c
int n = (int)read(…); // convert the next 4 bytes into an int
if( n > MAX_SIZE) // make sure the value is < MAX_SIZE
  error() // error if it’s not
strncpy(dest, source, n) // we’re safe to copy
```
Targeting file formats (or parsers)

• Signed bugs…
  • If \( n \) is a signed value (0x80000000 – 0xffffffff), the length check will be bypassed, and *huge* value will be read… overflow

• Parsers
  • Looping over data, expecting to find delimiters
    while(*ptr != ‘;’)
      *buff++ = *ptr++;

Types of file fuzzers

- **Brute force / mutation-based**
  - “dumb”
  - Given a set of good files, mutate them, and run them
  - Very fast and easy to start doing

- **Generation-based**
  - “intelligent”
  - Based off a very specific file format
  - Requires learning the format, and coding the fuzzer to fuzz specific fields following the format
  - 4 byte length, 2 byte version, string delimited by a ‘;’, etc…
  - Can be more fruitful, but requires more up-front effort
Popular file fuzzers

- Peach
  - De facto file fuzzer, very popular, very powerfull
  - [http://peachfuzzer.com/](http://peachfuzzer.com/)
- SPIKEfile
- notSPIKEfile
- iFUZZ
- etc.

- Each have their own pros and cons. We’ll be going into Peach in Part 2.
Pffft… but I have Python

• Tools are great… but we’re going to make our own ;)

Part 2 – Creating a mutation based file fuzzer, and a “how-to” for Peach file fuzzing.

https://github.com/rmadair/fuzzer

Feel free to get a head start and check it out. I highly encourage everyone to fork it, make changes, etc. 😊