Modifying Wine to Incorporate Forking

Parisha Joshi
parishamaheshj18@vt.edu
Virginia Tech

Deepthi Peri
pdeepthi@vt.edu
Virginia Tech

Yamini Gaur
yamini94@vt.edu
Virginia Tech

Alex Chiu
yves127@vt.edu
Virginia Tech

ABSTRACT

This paper aims to discuss the implementation of fork() capability on a Windows emulator like Wine. Since Windows and Linux do not share the same set of features, most of Linux-based improvements like fork() do not apply to Windows.In order to leverage the advantages of fork, this paper aims to modify WINE with a forkserver and report the results of fuzzing an application by using AFL fuzzer.

KEYWORDS

AFL, Wine, Fuzzing, Blackbox, Linux, WinAFL

1 INTRODUCTION

Fork() is employed by a large number of fuzzers to avoid having to waste time creating a new process for each input and to double the rate of fuzzing. Despite the popularity of Windows, fork() is still limited to Linux. This paper aims to emulate these capabilities on WINE and provide an exhaustive review of the methodology and the challenges faced in the process.We Implement a forkserver on AFL for Windows in order to increase the rate of fuzzing. Further, we test the fuzzer on an application on Wine and report the fuzzing time for the following scenarios:

- Fuzzing on Windows
- Fuzzing on Wine (modified AFL with forkserver)
- Fuzzing on Wine (unmodified AFL)

The rest of the paper is arranged as follows- Section 2 introduces Wine, forking as well as AFL fuzzer. Section 3 talks about the problem statement. Section 4 is about the proposed methodology and the steps carried out to modify WINE. Section 5 discusses the results obtained from the experiments and section **??** talks about how the results were evaluated and defines some imperative terminologies. Section 6 discusses the challenges faced during the experiment. Section 7 talks about the conclusion and future scope of the project.

2 BACKGROUND

2.1 Wine

The Windows operating system is the most popular operating system, beating Linux and Mac OS. But these Windows applications cannot be run on the other operating systems. This is where Wine serves its purpose. Wine (originally an acronym for "Wine Is Not an Emulator") is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems, such as Linux, macOS, BSD.It is a free software and the source code is available online. Instead of simulating internal Windows logic like a virtual machine or emulator, Wine translates Windows API calls into POSIX calls on-the-fly, eliminating the performance and memory penalties of other methods and allowing you to cleanly integrate Windows applications into your desktop.[6] There are over 26,000 Windows applications on the Wine application Database that can be run on Linux, macOS and BSD. Wine also includes its own versions of popular Windows applications like Notepad, Wordpad and Explorer. Wine provides Winelib, which allows its shared-object implementations of the Windows API to be used as actual libraries for a Unix program. This allows for Windows code to be bulit into native Unix executables. Since October 2010, Winelib also works on the ARM platform.

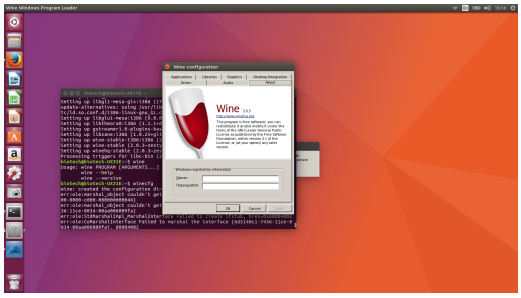

Figure 1: WINE in Ubuntu

2.2 Forking and ForkServers

Forking refers to an operation whereby a process creates a copy of itself. When developers take a copy of a source code from one software package, this operation will take place. The Child process will keep the content from the parent process, but it runs concurrently with the parent process but has its own address space. they use the same program counter, CPU registers and open files as the parent process. Forking is useful when the processor has to handle multiple requests. It accelerates the process of division of work.

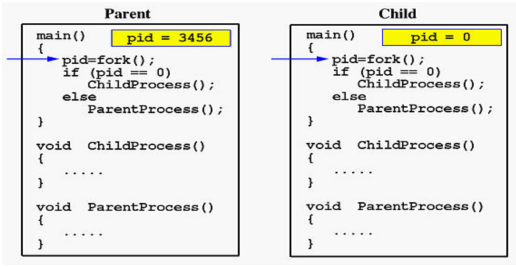

Figure 2: Forking Example

Figure 2 gives us an example of how forking works, we are having two processes running at the same time. One is the Parent Process, and the other one is the Child Process. When we are running the code, we use the fork() function to get the pid value. When the pid value is equal to 0, we are running the program in the Child process, otherwise, we are running the program in the Parent Process.
The biggest challenge for our project was Fork() command is only available in Linux. The fuzzer that we have chosen to use was American Fuzzy Lop, which use the fork() command to create multi-process while Fuzzing the program. However, Windows does not support the Fork() function. Instead, An alternative for Fork() in Windows is CreateProcess() , which is a rough equivalent of fork() + exec()The createProcess() function creates a new process, which runs independently of the creating process. [3]

2.3 CreateProcess()

Since Fork() command is only available in Linux. Windows does not support the Fork() function. therefore, we are using an alternative for Fork() command in Window, which is CreateProcess(). This command is a rough equivalent of fork() + exec() commands in Linux. The createProcess() function creates a new process and its primary thread, which runs independently of the creating process.The new process executes the specified executable file.
The CreateProcess function creates a new process and its primary thread. The new process executes the specified executable file. We use WinAFL to fuzz on Windows since WinAFL uses CreateProcess() to emulate the Forking in AFL If CreateProcess() succeeds, it returns a PROCESS_INFORMATION structure,containing handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle() function. [2]

2.4 Fuzzing: AFL

Fuzzing is a code testing technique, which consists of finding implementation bugs using malformed/semi-malformed information injection in an automatic fashion. Fuzzing is one of the most utilized strategies for automatic code testing. Through fuzzing, we can generate heaps of potential inputs for an application, consistent with a collection of rules, and inject them into a program to observe how the application behaves. Within the security realm, fuzzing is thought to be a good way to determine corner-case bugs and vulnerabilities. [1]

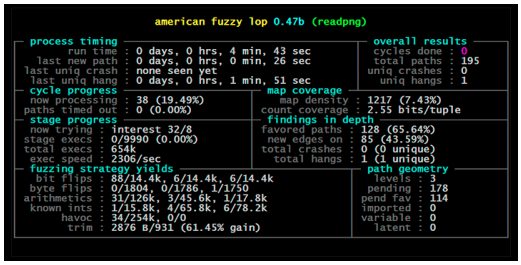

Figure 3: AFL Fuzzing in Linux

Most Fuzzing analysis is targeted on Linux and not on Windows. we use a fuzzer called the American Fuzzy Lop to carry out fuzzing, both on Wine still as Windows. American fuzzy lop is a security-oriented fuzzer that employs a completely unique type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states within the targeted binary. This considerably improves the functional coverage for the fuzzed code.Compared to alternative mainstream fuzzers, AFL offers a modest performance overhead and needs no configuration. It uses a brute force methodology and the fuzzing itself is administered by the AFL utility. AFL depends heavily on the Fork() system call to efficiently spawn many processes per second. this is one of the main reasons we chose AFL as our fuzzer.
Building the fuzzing environment is a crucial step. It not only involves taking care of installations but also includes finding the appropriate target to fuzz.
The steps to building the fuzzing environment are as follows:

- **Environment Preparation**: The prerequisites build essentials, latest clang and gcc can be installed using the following set of commands.

```
$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install clang
$ sudo apt-get install gcc
$ sudo apt-get install gdb
```

Figure 4: Commands to get the necessary build files, the latest gcc and clang [afl-steps]

- AFL can be installed from the Ubuntu repository or directly from the source. For the intent of this project, it was installed from the Ubuntu repo using the following command-

```
sudo apt-get install afl
```

Figure 5: Installing AFL from the Ubuntu repo [afl-steps]

- Optimization to reap performance benefits during the fuzzing process can be done by installing two clang wrappers- afl-clang-fast and afl-clang-fast++. These wrappers provide improvement in compile time speed during fuzzing.

```
$ cd ~/AFL/afl-2.36b/llvm_mode
$ LLVM_CONFIG=llvm-config-3.4 make
$ cd ../
$ make install
```

Figure 6: Installing clang wrappers for optimization [afl-steps]

- In order to perform fuzzing, the application's source code needs to be compiled using AFL's compiler wrappers. Let this be called the target-app. Afl-clang-fast compiler ensures that the binary is well instrumented.

```
$  CC="afl-clang-fast" CFLAGS="-fsanitize=address -ggdb" CXXFLAGS="-fsanitize=address -ggdb" ./conf
igure
$ make
```

Figure 7: Compiling with clang-fast [afl-steps]

- The next step is the actual fuzzing of the target application. The commands to do that are as follows. The first program is the general format for fuzzing and the second image is a more concise, application specific code snippet.

```
$ afl-fuzz -i [PATH TO TESTCASE_INPUT_DIR] -o [PATH TO OUTPUT_DIR] -- [PATH TO TARGET BINARY] [BINA
RY_PARAMS] @@
```

Figure 8: Format for running AFL [afl-steps]

```
$ cd ~/targets
$ afl-fuzz -i in/ -o out/ -- target-app_directory/target-app_binary @@
```

Figure 9: Format for running AFL [afl-steps]

The steps to fuzz an application using AFL fuzzer are:

- Compile a binary using compiler wrappers
- Fuzz the binary using afl-fuzz
- Review any unique crashes reported by afl-fuzz. Unique crashes occur if any of the afl-fuzz modified input files result in the target binary crashing.

The workflow of AFL is as follows:

- Afl-fuzz takes the test case file as the specified PATH input using the-i parameter and executes the target binary. It then monitors the binary activity for normal operation or crash and if no crash is detected, it stops the binary.
- Afl-fuzz then makes a minor modification to the initial test-case file and again executes the target binary using this new test-case file as input.
- Afl-fuzz takes a test case file as an input from the specified PATH using the-i parameter and executes the target binary, then monitor the binary activity for normal operation or crash, if no crash is detected, Afl-fuzz terminates the binary. Afl-fuzz then makes a minor modification to the initial test-case file and again executes the target binary using this new test-case file as input. [4]

2.5 WinAFL

The AFL fuzzer does not work on Windows since it does not support forking. WinAFL is a project that implements AFL using a different approach, that works on Windows, even for blackbox Fuzzing. Instead of instrumenting the code at Compile time, WinAFL used methods like Dynamic RIO and hardware tracing using Intel PT. Consequently, it offers different instrumentation modes. WinAFL executes multiple input samples without restarting the target program to improve the Process startup time. This is called the persistent fuzzing mode. [7]This is accomplished by selecting a target function (that the user wants to fuzz) and instrumenting it so that it runs in a loop. WinAFL uses CreateProcess() to emulate the functions of the fork() command in the AFL fuzzer, as shown in figure 10

```
if(!CreateProcess(NULL, cmd, NULL, NULL, inherit_handles, CREATE_SUSPENDED, NULL, NULL, &si, &pi)) {
  FATAL("CreateProcess failed, GLE=%d.\n", GetLastError());
}

child_handle = pi.hProcess;
child_thread_handle = pi.hThread;

if(mem_limit || cpu_aff) {
  if(!AssignProcessToJobObject(hJob, child_handle)) {
    FATAL("AssignProcessToJobObject failed, GLE=%d.\n", GetLastError());
  }
```

Figure 10: CreateProcess() in WinAFL

3 PROBLEM STATEMENT

Majority of fuzzing research is constrained to Linux-based systems. Despite the popularity of Windows systems, there are a lot of differences between Windows and Linux OS. Linux leverages fork() to increase the speed of fuzzing and to avoid wasting time creating a new process for each input. As discussed before, forking spawns a new process from the point of call from the parent process. This creates a divergence between the child and the parent process that can be handled via copy-on-write. Windows unfortunately does not have provision for fork().
The problem statement is to modify windows applications and programs with a forkserver in order to increase the rate of fuzzing using a windows emulator like WINE.

4 METHODOLOGY

In order to modify Windows applications using forkserver on Wine, we carried out the following experiments.

4.1 Experiments

Fuzzing with afl initially gives check-crash-handling() error.User can simply type "sudo bash" and modify the /proc/sys/kernel/core-pattern directory as suggested on the screen. Follow the instructions and we are good to go.

The first attempt to fuzz the application with wine on "audacity". The source code of which can be found from references.There are versions of it available for Linux and Mac Os platforms, but to serve the purpose of wine we made use of windows .exe file.The command is" afl-fuzz -Q -i /in -o /out -M wine audacity-win-2.3.1.exe", Which gives check-binary() error. Since WINE is just a shell and "wine application.exe" is a whole process the afl yet does not know where

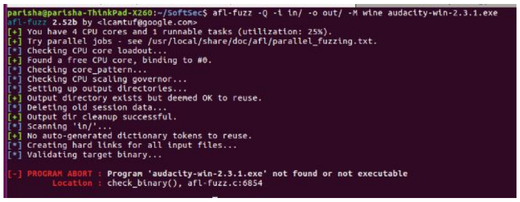

Figure 11: afl attempt on audacity.exe

it is supposed to inject the input cases.Here -Q is an instrumentation tool named QEMU for black box fuzzing provided in afl package itself. Removing -M from the command gives the error "fork server

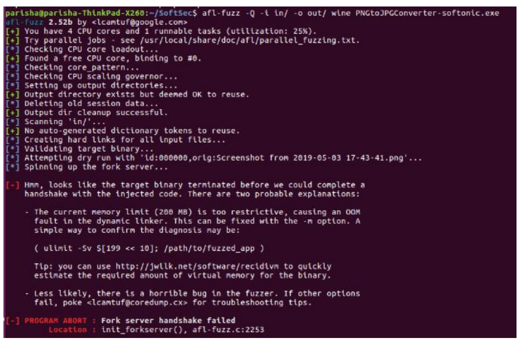

Figure 12: error without memory located to child process

handshake failed" as shown in Figure 12.

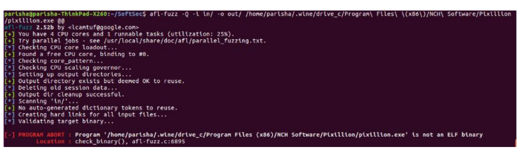

Figure 13: WINE in Ubuntu

As another attempt to try and fuzz the .exe file from the wine directory system we tried the command "afl-fuzz -i /in -o /out /path/to/.wine/drive-c/program files/application.exe @@". WINE creates it's own drive to store program files as if it is imitating windows file system but as it still is not an ELF binary The attempt is served by an error(Figure 13). We also tried to find ways to

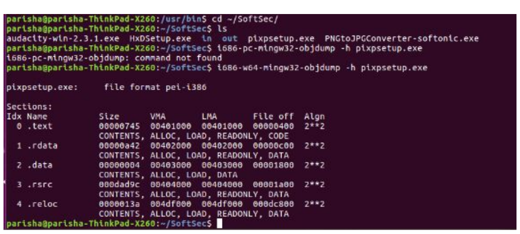

Figure 14: .exe to ELF conversion-1

convert .exe file into binary files. using following method. We used

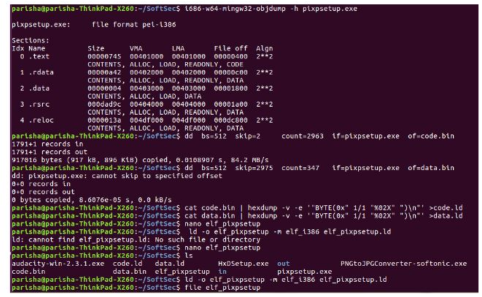

Figure 15: .exe to ELF conversion-2

the mingw cross compiler to dump the section headers from .exe file followed by extracting the raw section data from .exe file.It is an interesting yet time consuming approach and because of it's uncertainty we did not go ahead with it.

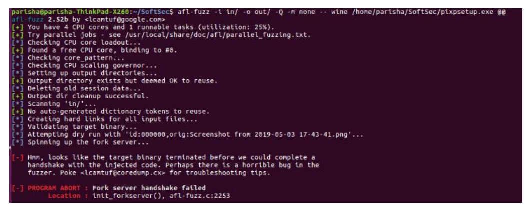

Figure 16: afl-fuzz with "@@"

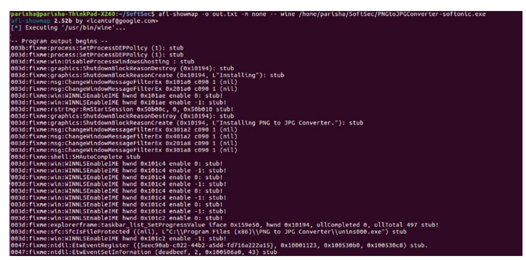

Figure 17: afl-showmap

Figure 16 and 17 both shows the error messages received when using afl-fuzz and afl-showmap.It is a tool that shows target binary content bitmap in human readable form. In attempt to fuzz the WINE itself the input directory contained .exe file. The smallest we could find was 130KB in size,which is very huge for a fuzzer. Running the fuzzing for almost an hour could not complete one cycle and only one new path got detected.Notice the -Q instrument in the command which shows the use of QEMU mode of AFL.[5] It is an emulator which is used to fuzz the blackbox or closed compiled binaries. To use this feature user has to compile the qemu-mode directory under afl folder. It takes longer than usual(2-5 times)fuzzing time for the implementation but is better than DynamoRio tool in winAFL.

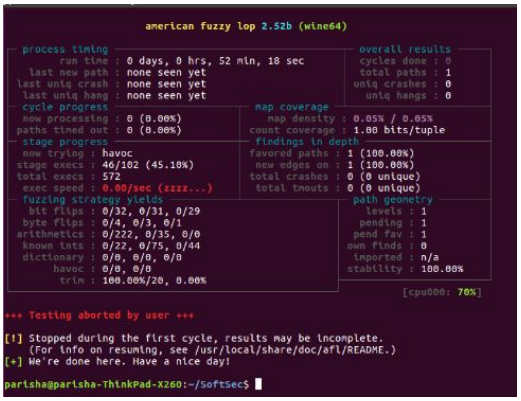

Figure 18: Fuzzing of WINE64

4.2 Implementation

We implemented afl-fuzzing on the WINE64 without specifying the application in the command line. The applications are given as input to the wine program in /in directory. The afl flips bits for it's tests in the .exe file which is 130kb long and checks for all the outputs. The reason why the fuzzing is slow is clear now.For generating a crash suppose there exists at least one combination of bits of the input file. If the file is too large, reaching to that combination would take a lot of time.

5 RESULTS

We experimented with a lot of windows and linux applications to play around afl and QEMU mode but the most significant results can be shown from two applications below. The resulting output directory gives three folders named: crashes, queue and hangs. Because the fuzzing with wine could not find any crashes in given time all these folders were null.Additionally fuzzer-stat and plot-data text files gave the summary of execution data. The fuzzer-stat file results are in the Figure 19.

For comparison with wine application we fuzzed an application called binutils-2.32. Which takes an input of elf file and also gives slow fuzzing results.This time, because binutils is not a windows application WINE is not necessary.The statistics are given below.(Figure 19,20,21)

Although the afl was only run for 12 minutes on binutils application, the execution speed is quite high.The number of executions done are 1344 times the number of execs done on WINE64. bitmap coverage is almost 400 percent more than previous.544 paths are found in 12 minutes and the target mode is set to default(afl-fuzz).

6 CHALLENGES AND DRAWBACKS

The challenges faced in implementing this project are as follows-

- Modifying Windows applications with a forkserver is challenging. CreateProcess() and CreateThread() commands are close to the functionality of fork() in Linux but are not identical to it. This resulted in a steep learning curve for understanding these concepts and using them with AFL.
- Finding out the target file to fuzz was challenging. Our initial initiative was to try to fuzz any .c file and test AFL to remove command line bugs and errors. However, choice of target file with AFL that shows successful fuzzing is a crucial task.
- Choice of executable files that could be fuzzed using AFL was a hurdle. It had to be ensured that the application was available on WINE and was simple to fuzz.

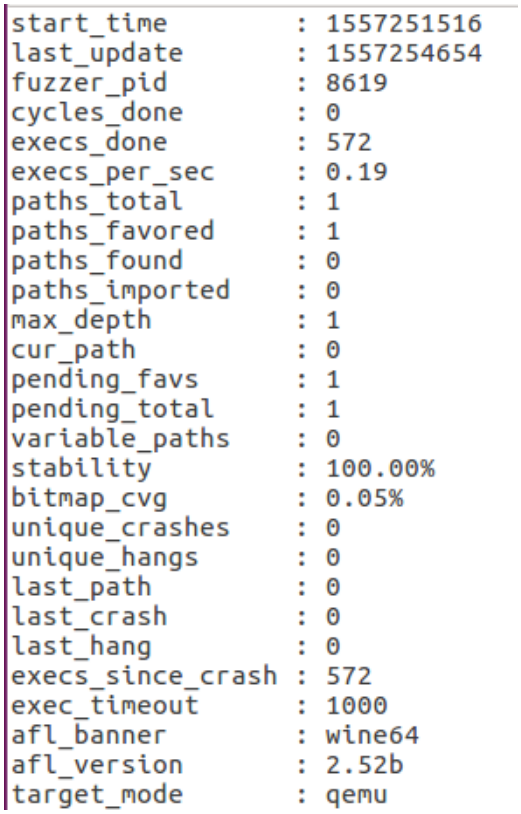

Figure 19: Resuls of fuzzer-data:WINE64

Figure 20: Resuls of fuzzer-data: binutils-2.32

7 CONCLUSION AND FUTURE WORK

In conclusion, we carried out experiments to fuzz Windows applications on Wine. We made use of AFL and used CreateProcess() to emulate the functionality of forkserver. Since forking is not supported on Windows and doubles the rate of fuzzing, it proves to be

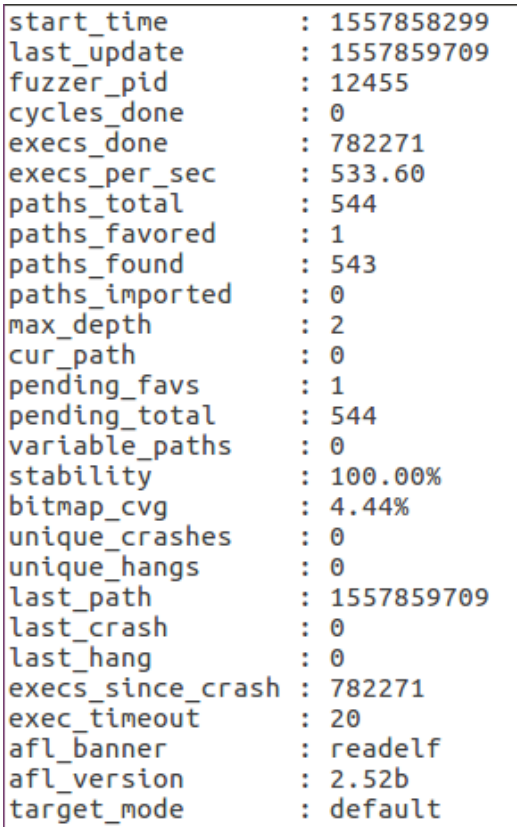

Figure 21: Resuls of fuzzer-data: binutils-2.32

an interesting concept for security research. As aforementioned, we were able to successfully carry out fuzzing of applications on Wine using AFL.

The applications fuzzed for the scope of this project were simple file converters and pdf readers. In the future, this project could be extended to more complex and extensive applications

Microsoft recently released the news that the new Windows 10 will be shipped with a Linux kernel. Since CreateProcess() does not emulate forking completely, this would eliminate the dichotomy between Windows and Linux when it comes to fuzzing.

REFERENCES

[1] Radu-Emanuel Chiscariu. "How to Use Fuzzing in Security Research". In: (). URL: https://www.ixiacom.com/company/blog/how-use-fuzzing-security-research.
[2] "CreateProcess()". In: (). URL: https://docs.microsoft.com/en-us/windows/desktop/ProcThread/creating-processes.
[3] "Forking in C". In: (). URL: https://www.geeksforgeeks.org/fork-system-call/.
[4] "Hunting For Bugs With AFL 101 - A PRIMER". In: (). URL: https://research.aurainfosec.io/hunting-for-bugs-101.
[5] "Introduction to QEMU". In: (). URL: http://www.linuxfromscratch.org/blfs/view/systemd/postlfs/qemu.html.
[6] "Introduction to Wine". In: (). URL: https://www.winehq.org/.
[7] "WinAFL tips and Tricks". In: (). URL: https://blog.fadyothman.com/afl-tips-tricks/.