FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

In-Memory Protocol Fuzz Testing using the Pin Toolkit

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Patrik Fimml
Registration Number 1027027

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dipl.-Ing. Markus Kammerstetter

Vienna, 5th March, 2015

Patrik Fimml   Markus Kammerstetter

Erklärung zur Verfassung der Arbeit

Patrik Fimml
Kaiserstraße 92
1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. März 2015

Patrik Fimml

Abstract

In this Bachelor's thesis, we explore in-memory fuzz testing of TCP server binaries. We use the Pin instrumentation framework to inject faults directly into application buffers and take snapshots of the program state. On a simple testing binary, this results in a 2.7× speedup compared to a naive fuzz testing implementation. To evaluate our implementation, we apply it to a VNC server binary as an example for a real-world application. We describe obstacles that we had to tackle during development and point out limitations of our approach.

Contents

CHAPTER 1

Introduction

Fuzz testing is a well-established technique for finding bugs in programs and can be applied even without access to the source code. This makes it a useful tool for security researchers to identify interesting parts of a program.

While it is easy to write a simple fuzzer, there is a computational overhead attached to how the fuzzed data enters the application. In our case, we focus on server applications.

In-memory fuzz testing is a technique that allows us to reduce this overhead. We develop an approach to modify buffers directly in application memory. We take a look at the Pin framework, which allows us to dynamically instrument application binaries, and use it to take snapshots of the program state.

Finally, we evaluate our proof-of-concept tool on real-world binaries, and perform a performance comparison with a more traditional protocol fuzz testing approach.

CHAPTER 2

Fuzz Testing

Fuzz testing or *fuzzing* is a technique that attempts to find bugs and security vulnerabilities in applications by providing invalid input [Oeh05; Wan11]. Typically, such input is generated randomly. When a particular input causes the program to hang or crash, this indicates a potential bug [Wan11]. These inputs are then recorded for detailed manual inspection.

Sophisticated techniques have been developed to choose such program inputs. In this work, we focus on mutating previously captured messages randomly, which is the simplest method to generate test data.

To perform fuzz testing on a program, we need to be able to continuously execute a small region of it. In particular, we want to repeatedly run methods that deal with parsing the application protocol. In each iteration, we want to use a slightly different argument.

2.1 Naive Protocol Fuzz Testing

A naive approach is to initiate connections sequentially. While simple and somewhat robust, there are some drawbacks to this approach:

TCP overhead: Lots of connections are established and then torn down.

Protocol overhead: If the program needs a long sequence of data exchange initially to get to the interesting region, this initial data exchange has to be performed for every new connection.

Side effects: If the program crashes e. g. due to a memory corruption vulnerability, the actual cause might have come from a previous connection.

Specifically, the overhead is high when only a small part of the program should be fuzzed, but the initial sequence to get to this point in the program is long in comparison.

2.2 In-Memory Fuzz Testing

Instead of executing the program completely with all network handling code for each iteration, *in-memory fuzz testing* limits execution to the parsing routine. This can be achieved by taking a snapshot of the program state, injecting a fault, and restoring it later for the next iteration [Sut07, p. 42].

Instead of sending packets, we directly modify the application's buffer in memory and repeatedly execute the parsing code on it. The modified control flow is illustrated in Figure 2.1.

In most cases, the function under test will modify some state variables, so that simply executing the function repeatedly will not provide the desired effect. We need to save the program's state upon entry to the region to be fuzzed. Later, upon leaving the interesting region, we can reset the program state to the snapshot we took before.

There are several ways to capture the program state. In our approach, we instrument all memory writes in the program so that we can save memory contents before they are modified. This is somewhat naive, as all other aspects of the program state are not captured. In particular, system calls are not intercepted, thus any side-effects on file handles or memory management will leak between iterations and potentially interfere with the operation of the fuzzer, leading to false positives.

More sophisticated possibilities to take snapshots include the following:

- Use a virtual machine to take snapshots of not only the program, but the whole operating system. This is the most transparent method, but also potentially incurs high overhead and might even be more costly than a naive fuzz testing tool with a new connection for each attempt.
- Use `fork()` to create a copy of the process with its own memory space for each new iteration. This has the advantage that the Linux kernel implements `fork()` with copy-on-write semantics, thus potentially reducing overhead. Shared file descriptors could still pose a problem, and the operation is not transparent to the fuzzed process. Coordinating processes would be more complex in the tool as well.

The snapshotting mechanism employed by our tool is described in detail in Section 4.1.

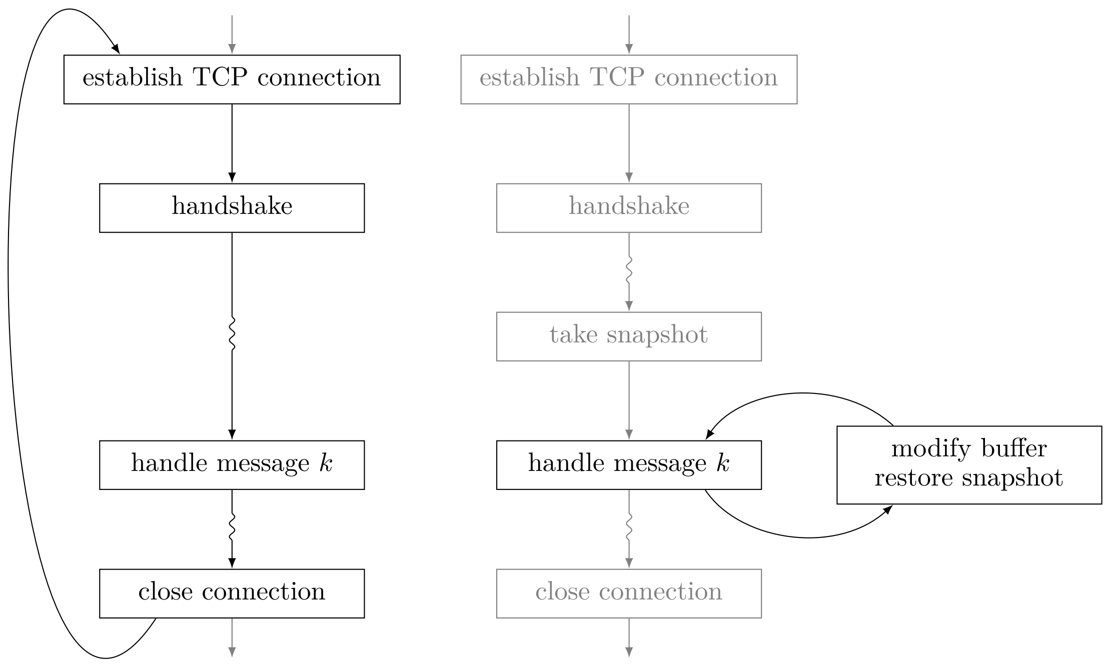


Figure 2.1: High-level overview of the control flow in the naive (left) and in the in-memory protocol fuzz testing approach (right). In this example, we are interested in testing the code for a particular message k. In the naive approach, connection setup and initial communication are repeated for each iteration. With in-memory fuzz testing, this is only done in the first iteration. In subsequent iterations, only the interesting program region is executed.

CHAPTER 3

The Pin Framework

Pin [Luk05] is a framework for dynamic binary instrumentation, i. e. the analysis and modification of programs at runtime. The framework is developed by Intel.

While the software is proprietary, it is available free of charge for non-commercial purposes and can be obtained from the Intel website [PinWeb].

Dynamic instrumentation has a range of applications. It enables analyses that cannot feasibly be performed statically. Simple analyses might gather statistics about program execution (profiling). However, instrumentation also allows for more complex inspection of a program, such as memory sanity checking.

The Pin framework stands out due to its versatility and its aim for good performance.

3.1 Terminology and Concepts

Pin is the name of the instrumentation framework. `pin` is also the name of the program that one runs to executes a target binary under the supervision of the Pin framework.

A *Pintool* is a shared library that the Pin framework can load and execute. Through calls to the Pin API, Pintools define how the target application is instrumented and provide the actual functionality of interest.

To do this, a Pintool can use the API to register so-called *instrumentation functions*. Instrumentation functions can be registered on multiple levels, e. g. on the routine (RTN), basic block (BBL) or instruction (INS) level, depending on how fine-grained the instrumentation needs to be.

When running the target binary, Pin will perform *just-in-time compilation* (JIT) to instrument it. This means that whenever a certain region of the program is called for the first time, it re-compiles the original code, inserting instrumentation code and possibly changing register allocations along the way. Future calls to the program region will directly use the compiled block. The mechanism and optimizations are described in [Luk05].

To determine how instrumentation should be performed, the JIT compiler calls the instrumentation functions registered by the Pintool. The instrumentation function can then choose to inject code into the binary. In Pin terminology, code injected this way is called an *analysis function*.

While analysis functions are defined as regular C functions in the Pintool, inserting them does not automatically mean that there will be a function call. The JIT compiler will analyze the binary and can inline simple analysis functions into the target application's sequence of instructions. This means that the overhead from adding instrumentation can be kept low.

Since the code in analysis functions is run every time a certain program region executes, but code in the instrumentation function is only run once per program region, it is advisable to shift computations and branches to the instrumentation function whenever possible [PinGuide, Section "Performance Considerations"].

CHAPTER 4

Implementation

4.1 Snapshotting Mechanism

To provide the ability to snapshot the program state and restore it later, we use Pin to instrument all memory writes in the program. For each memory write, we store the address and the previous value at that address, so that we can later restore the original memory contents and effectively undo the modifications.

This functionality is implemented in `snapshot.cpp`. We define a structure to store information about each memory write that takes place, shown in Listing 4.1.

We assume that most memory writes will be 64 bit wide. Operations with memory operands that are 8 byte or smaller in size are stored directly in one entry. For memory writes larger than 8 bytes, we store multiple 8-byte entries.

To enable Pin to inline our analysis functions, we want to keep them simple as possible. To this effect, we define separate functions for each memory operand size. The analysis function for memory operands of 8 bytes or smaller is shown in Listing 4.2.

Later, we use *INS_InsertPredicatedCall* to instrument all memory write instructions with these functions, as shown in Listing 4.3.

By keeping logic in the instrumentation method, we reduce overhead. Another possible optimization would be to eliminate the call to *NextMemwriteTrace.* In most cases, this method only needs to return the next available slot. However, it might be necessary to grow the buffer. One could use *INS_InsertIfCall* and *INS_InsertThenCall* to inline this check and eliminate the overhead of a function call in most cases.

```
struct memwrite_trace {
    UINT64 addr;
    UINT64 val;
    UINT8 len;
};
```

Listing 4.1: Definition of the structure to record memory writes. *addr* stores the memory write address, *val* stores the previous value at that address, and *len* indicates the size of the memory write in bytes. Memory operands of instructions can vary in size (e. g. store byte/store word).

```
#define DEFINE_OnMemoryWriteA(n) \
    static void OnMemoryWrite ## n (ADDRINT addr) { \
        struct memwrite_trace* trace = NextMemwriteTrace(); \
        trace->addr = addr; \
        PIN_SafeCopy(&trace->val, (void*)addr, n); \
        trace->len = n; \
    }
```

Listing 4.2: Analysis function for memory operands that are 8 bytes or smaller in size.

```
static VOID Instruction(INS ins, VOID* unused) {
    if (INS_IsMemoryWrite(ins)) {
        for (UINT32 i = 0; i < INS_MemoryOperandCount(ins); i++) {
            if (INS_MemoryOperandIsWritten(ins, i)) {
                AFUNPTR fun(nullptr);
                switch (INS_MemoryOperandSize(ins, i)) {
                case 1: fun = AFUNPTR(OnMemoryWrite1); break;
                case 2: fun = AFUNPTR(OnMemoryWrite2); break;
                case 4: fun = AFUNPTR(OnMemoryWrite4); break;
                case 8: fun = AFUNPTR(OnMemoryWrite8); break;
                case 16: fun = AFUNPTR(OnMemoryWrite16); break;

                default:
                    fprintf(stderr,
                            "pinfuzz:␣not␣implemented:␣memory␣operand␣of␣" \
                            "size␣%d\n",
                            INS_MemoryOperandSize(ins, i));
                    PIN_ExitProcess(1);
                }
                INS_InsertPredicatedCall(ins,
                                         IPOINT_BEFORE,
                                         fun,
                                         IARG_MEMORYWRITE_EA,
                                         IARG_END);
            }
        }
    }
}
```

Listing 4.3: The instrumentation function that enables memory snapshotting.

4.2 Snapshots and Parallelism

Server applications may spawn multiple processes or threads to handle a request. Frequently, a new thread or process is spawned when a new connection is established. The child thread or process is then responsible for handling communication on this connection.

In more complex applications, communication between different threads might be required to process a request. In simple multithreaded applications where the connection-handling thread does not communicate with other threads, it might be possible to modify the application binary to keep running in a single thread, e. g. by patching out a `fork()` call. Nevertheless, it is desirable to be able to use the application binary as-is without requiring manual modifications.

Our in-memory fuzz testing approach uses snapshots to reset application threads to a state reached at an earlier point in time. When multiple threads communicate with each other, we will generally need to snapshot and reset not only the thread handling the connection, but all application threads. Moreover, as communication might also take place across process boundaries, we need a way to snapshot all application threads across all processes.

Pin provides a set of functions for stopping other threads and examining their current register values. `PIN_StopApplicationThreads` can be used to stop application threads. However, each of the application threads finishes the currently-running *trace* first, i. e. threads are only stopped on the next branch or return instruction. This means that if a thread is executing a blocking system call, `PIN_StopApplicationThreads` might block indefinitely. While this might be acceptable for some computationally heavy applications that do not interact with the operating system frequently, it is not very well-suited for our use case.

We also need a way to handle the case where an application spawns multiple processes. Following child processes is rather easy with Pin, and it provides a way to register hook functions for the `fork()` system call. However, facilities for inter-process communication are not provided by Pin and have to be implemented.

Multiple options are available for implementing this. TCP or Unix-domain local sockets are straightforward, but not well-suited since we would like to measure the overhead of socket communication as well. Pipes as well as sockets become complex to deal with as soon as more than two processes are involved, since some way of broadcasting information to other processes is required. Instead, we establish a shared memory area which can be accessed from all processes. While this should result in a low overhead and allows for broadcasts to all listeners, it has its drawbacks as well. Most notably, the logic becomes more complex and synchronization between threads is non-trivial.

It turns out that introducing a per-process *control thread* simplifies the control flow a bit. An example is shown in Figure 4.1. Control threads are *internal threads* in Pin terminology, as they exist independently of any application threads. A hook is installed to spawn a new control thread upon process creation.

The control thread collects information about the application threads in its own process,

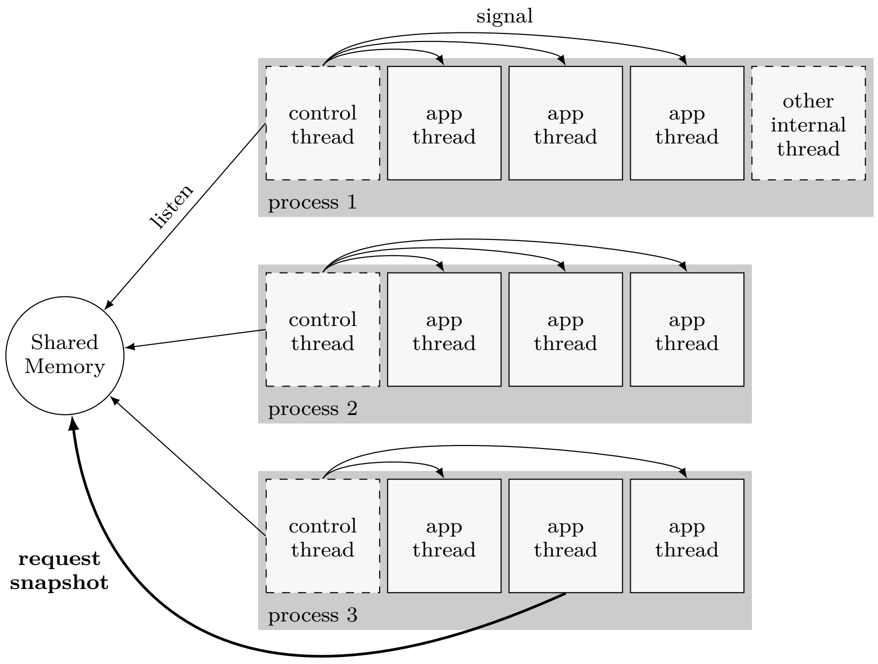


Figure 4.1: Snapshotting mechanism for multi-threaded, multi-process applications. A thread hits a breakpoint and sends a request to the control threads using the shared memory area. In each process, the control thread receives the request and signals the application threads in the process (except for the thread that initiated the request). When all threads have switched context to the signal handler and no application code is running anymore, the request is carried out. After completion, control returns to the application code.

such as the number of application threads and their thread IDs. With this information, the control thread then handles requests from other processes and signals the application threads in its own process.

The advantage of this approach over one where any thread can signal all other threads across processes directly is that management of the shared memory area becomes easier. Instead of having to resize the shared memory area or needing to ensure that it is big enough from the beginning to be able to hold information about all application threads, this information is stored per-process, where memory management is an easy task. The shared memory area has a fixed size which is sufficient for communication with control threads in an arbitrary number of processes.

Once the control thread has received a request, it needs a way of interrupting application threads and running code from the Pintool. Under Linux, we use the `tgkill` system call to send a `SIGUSR1` signal to a specific thread. The signal is caught by a previously installed handler, which then performs the desired action.

Under Linux, signals can interrupt system calls, so this approach will work even when the application thread is in the middle of an operation. This comes with some caveats, however. When the application resumes with unchanged context, the system call will return an errno with `errno` set to `EINTR`: Interrupted system call. Depending on how the application handles `EINTR`, this can lead to unexpected results. This could be improved in the future.

For Windows, no interruption mechanism has been implemented yet. Potentially, Asynchronous Procedure Calls (APCs) or debugging functionality in the WinAPI could be used to implement such a facility. As of now, only single-threaded processes are handled correctly under Windows.

4.3 Address Expressions

For our Pintool to be able to modify a buffer in the application, we need to know its address and its length. The application buffer might be located either in the data segment, on the heap or on the stack. In general, its address cannot be predicted and it might be necessary to examine memory at runtime to determine its location.

In order to avoid having to recompile the Pintool for different applications, all aspects of the operation of our tool need to be configurable. Hence, we need a way to dynamically specify the address of the message buffer.

Since we might not know the address in advance, we want to allow the address to depend on memory and register contents. In other words, we would like to be able to evaluate simple mathematical expressions depending on registers and memory contents.

Several alternatives were considered to implement this. The *Lua* scripting language seemed like a good candidate, since it is lightweight and designed to be embedded in applications. However, Lua stores all values as double-precision floating point values internally. Since this is problematic when doing calculations with 64-bit addresses, Lua was no viable option.

The *Python* scripting language can also be embedded in applications, but it is more heavy-weight with a considerably larger runtime.

Hence, a simple custom expression parser was implemented using the standard tools `yacc` and `bison`. The supported syntax is shown in Table 4.1.

The simple language supports basic arithmetic on its operands, which can be integer literals or values coming from registers. Register names are looked up in the Pin register table dynamically, resulting in platform-independent code.

As an additional operator, unary `*` is provided as a C-like memory dereference operator. This operator returns the value at the memory address specified by its argument, by accessing application memory using `PIN_SafeCopy`. The expression parser represents all values as 64-bit integers internally and memory reads fetch a 64-bit value from memory. This might be adapted to better suit 32-bit systems in the future.

With this expression language, the specification of buffer addresses becomes rather easy. A buffer on the stack might be specified as `rbp - 0x40`, or a buffer whose address is stored in a stack variable could be referred to as `*(rbp - 12)`.

N...	Decimal literal, e. g. `0`, `42`
`0x`*N...*	Hexadecimal literal, e. g. `0x1`, `0xFF`
register	Register access by name, e. g. `eax`, `rbp`
expr `+` *expr*	Add
expr `-` *expr*	Subtract
expr `*` *expr*	Multiply
expr `/` *expr*	Integer divide
`(`*expr*`)`	Grouping
`*`*expr*	Memory dereference (64-bit)

Table 4.1: Syntax for address expressions in the configuration file. Every item listed here is a valid subexpression, referred to as *expr*.

4.4 Crash Handling

Whenever the program under test produces a crash, we want to record the location as well as the cause of the crash, and return the application to a functional state. Typically, crashes are caused by specific inputs sent by the fuzz testing tool, so that restoring memory contents is sufficient to recover from a crash.

To do this, Pin allows us to register *context change callbacks*. We can use these to intercept signals on Linux and Exceptions on Windows. From the callback, we can inspect and modify the program state. Under Linux, this is not sufficient since the context change callback is only called after the program has already terminated. Thus, we specifically intercept the `SIGSEGV` signal to intercept segmentation faults before they lead to program termination.

4.5 Reporting Exceptional Application Behaviour

Whenever a certain packet seems to make the application behave in an unexpected way, we want to record the precise circumstances and the packet that caused the unexpected behaviour. In particular, we can catch the following situations:

- When the application crashes, in particular when it produces a segmentation fault.
- When the application does not reach the end of the specified program region within a specified amount of time.

As soon as one of these situations occur, information about the situation is written to the logfile. In addition, we need to record the packet that triggered the failure. We chose to write a packet capture file in the *Pcap* file format as tools are readily available for this file format.

Since only the actual payload is of interest, Pcap files written by the tool have fake IP and TCP headers to keep the implementation simple.

CHAPTER 5

Usage Instructions

A few preparatory steps are necessary to use our fuzz testing tool. The environment needs to be set up to build the tool and information about the target binary needs to be provided in a configuration file. These steps are described in this chapter.

5.1 Setup on Linux

On Linux, compilation is pretty straightforward. Most distributions ship the dependencies in packages. On an Ubuntu system, apart from standard development packages, the following packages are required:

- `libboost-chrono-dev`
- `libboost-dev`
- `libpcap0.8-dev`

By default, the Clang++ compiler based on the LLVM framework is used to compile the Pintool. Clang supports thread safety annotations, which are used in the code to guard against thread programming erros. However, the tool can also be compiled using GCC. The build process is invoked by executing `make` in the `src` subdirectory.

5.2 Setup on Windows

In the following sections, we describe the process to build the Pintool on Windows. We used a Windows XP instance for testing.

The following software packages are required to build the tool on Windows:

- Pin
- Microsoft Visual C++
- MinGW
- MSYS
- Boost
- WinPCap

MinGW (Minimalist GNU for Windows) and MSYS provide a basic set of Unix tools that are required in the build process, most notably GNU `make`. Installation is straightforward with the provided installer.

After having MinGW and MSYS installed, we need to include their `bin/` directories in our `PATH`. Since we want to use the toolchain from Visual C++, we need to include these binaries in our `PATH` as well. We can use the `vcvarsall.bat` script provided by Visual C++ for this.

The commands to set up environment variables can be seen in Listing 5.1.

Some parts of Boost need to be compiled in order to use them, so we need to build them as well, as shown in Listing 5.2.

Pin and WinPCap ship with binaries included, so no compilation steps are needed for them.

Once all prerequisites are in place, the build process can be started by executing `make` in the `src` subdirectory.

```
PATH=C:\MinGW\bin;C:\MinGW\MSYS\1.0\bin;%PATH%
call "C:\Programme\Microsoft␣Visual␣Studio␣10.0\vc\vcvarsall.bat" x86
```

Listing 5.1: Commands to prepare the environment for building a pintool on Windows.

```
cd third-party\boost_1_57_0
bootstrap
b2 --build-type=complete --with-system
```

Listing 5.2: Compiling Boost under Windows.

5.3 Target Binary Analysis

Some knowledge about the target binary is required before fuzz testing can be performed. Typically, one will perform basic analysis of the binary with a disassembler. As part of this analysis, the following items need to be identified:

- how the application accepts connections,
- how the application processes messages.

In particular, one needs to identify the region of the program code that should be subject to fuzz testing. The tool will loop over this region in every iteration, mutating the memory buffer as specified.

The structure of a typical program and where breakpoints need to be set is shown in Listing 5.3. Breakpoints are described in more detail in Section 5.4.

In addition to the binary analysis, a packet capture has to be prepared that contains a regular request to the target binary that will then be replayed. The packet capture needs to be in `pcap` format and can be produced using tools such as `tcpdump` or Wireshark.

```
                      server_loop() {
                          server = socket(...);
                          bind(server, ...);
                          listen(server, ...);
     bkpt_listening→      ...
                          while (1) {
                              accept(server, ...);
   bkpt_established→          client = socket returned by accept;
                              ...
                              client_handler(client);
                          }
                      }

                      client_handler(client) {
                          while (1) {
    bkpt_region_end→          recv(client, ...); // read header and length
                              recv(client, ...); // read complete message
  bkpt_region_start→          ...
                              process_message(...);
                              if (error) {
    bkpt_region_end→              close(client);
                                  return;
                              }
                          }
                      }
```

Listing 5.3: Typical structure of a server program, illustrating how breakpoints should be placed. We set `bkpt_listening` and `bkpt_established` to instructions *after* the socket has been set up for listening or a connection has been accepted, respectively. `bkpt_region_start` is set on an instruction *after* the full message has been received. `bkpt_region_end` is set before the next message is read. We also set it on the error handler before the socket is closed.

5.4 Configuration

Parameters controlling tool operations are specified in a configuration file in the `INI` file format. An example configuration file is shown in Listing 5.4.

```
[FuzzConfig]
pcap = pcap001.pcap
logfile = pinfuzz.log

[pinfuzz]
sequence = connect, 4, r, A, 6, B, r, 8, r, 10, r, 12, r, 14, r
port = 2000
bkpt_listening = 0x400d84
bkpt_established = 0x400a98
bkpt_region_start = 0x400ae3
bkpt_region_end = 0x400b24, 0x400b99
regionfuzz_buffer = rbp - 0x40

[Packet4]
range = 8-10

[Packet6]
nofuzz = True
```

Listing 5.4: Example configuration file for our tool.

The following configuration parameters are defined:

[FuzzConfig] section

pcap=
The packet capture file to read packets from.

logfile=
The logfile where information about crashes is written to.

[pinfuzz] section

sequence=
The comma-separated sequence of actions to be performed by the fuzzer. Actions can be:

connect	Open a TCP connection to the server.
A	Start of the in-memory fuzz testing region.
B	End of the in-memory fuzz testing region.
N	Send packet number *N* from the pcap file.
r	Receive a packet from the server.

port=
The TCP port to connect to.

Breakpoints: Breakpoints are set by specifying an instruction address, and are typically specified in hexadecimal notation. All breakpoints apply *just before* the instruction at the given address is executed. See also the example in Listing 5.3.

bkpt_listening=
Breakpoint where the application is listening for incoming connections.

bkpt_established=
Breakpoint after the connection has been accepted.

bkpt_region_start=
Start of the program region to be fuzzed.

bkpt_region_end=
End of the program region to be fuzzed.

skip_region_start=
How many crossings of *bkpt_region_start* should be skipped. Defaults to 0.

regionfuzz_buffer=
Specifies the memory location of the buffer to be fuzzed. This is an expression that can access register contents and dereference memory locations, as described in Section 4.3. The expression is evaluated at the point specified by *bkpt_region_start*.

[PacketN] section

This section may be present multiple times. *N* in the section name is the index of the packet that it applies to.

nofuzz=True
Do not change any bytes in this packet.

range=
The range of bytes that can be fuzzed, 0-based. e. g. *0–3,12*

5.5 Running the Tool

When the Pintool has been compiled as a shared library file, a target executable has been selected and a configuration file has been created, you can start the fuzz testing process. The command line for Linux and Windows is shown in Listings 5.5 and 5.6 respectively.

```
third-party/pin-linux/pin
       -tool_exit_timeout -1
       -t src/obj/pinfuzz.so
       -c CONFIGFILE -- CMDLINE...
```

Listing 5.5: Linux command line.

```
third-party\pin-windows\pin
       -t src\obj-win\pinfuzz.dll
       -c CONFIGFILE -- CMDLINE...
```

Listing 5.6: Windows command line.

The `CONFIGFILE` parameter specifies the path to a Pinfuzz configuration file as described in Section 5.4. `CMDLINE` is the command line that you would normally use to run the program under test, i. e. the path to the executable and any arguments.

In multi-process applications, the fuzz testing control logic is handled inside a thread in the main process. When this process terminates, Pin would kill our control thread after some time. This is why we use the `-tool_exit_timeout` parameter on Linux. On Windows, only single-process, single-threaded targets are supported.

CHAPTER 6

Evaluation

6.1 Experiments with VNC Server

In order to test our approach on real-world software, we picked a VNC server application on which to perform fuzz testing. The VNC protocol was chosen as a target since it is a binary protocol with a relatively simple structure.

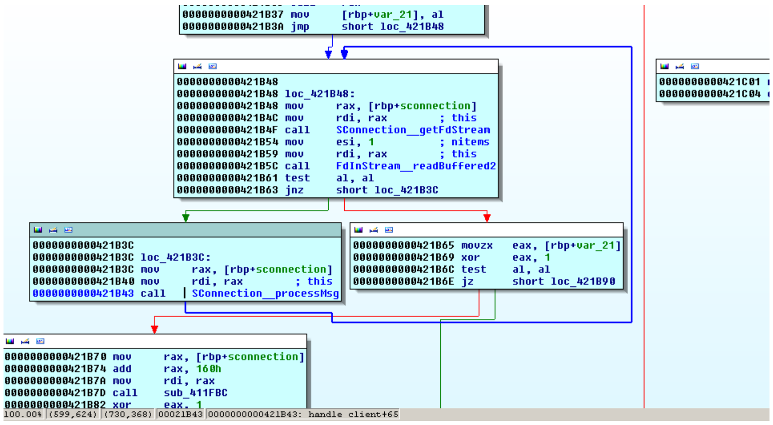
For our experiments, we used the `x0vnc4server` binary from the `vnc4server` Ubuntu package, version `4.1.1+xorg4.3.0-37ubuntu5.0.1`. This is the open source version of the VNC software distributed by RealVNC [RealVNC].

The `x0vnc4server` program listens on port 5900 for incoming connections. When a connection is established, client and server perform protocol negotiation and authentication. Following this, the VNC server connects to an already-running X server and shares the display contents over the VNC connection.

We analyzed the binary without looking at the source code to simulate a black-box binary analysis situation. Some reverse engineering effort was needed to find out how the program processes incoming network connections.

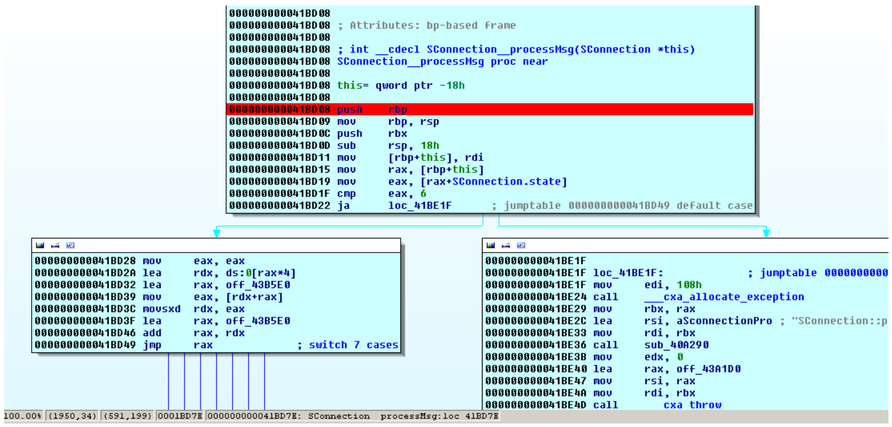
The core functionality for reading from the network connection is shown in Figure 6.1. An `SConnection` object is created to handle incoming connections. The `SConnection` object has an instance of `FdInStream`, which handles reading and buffering of messages. By examining the `FdInStream` functionality, we found where the message buffer is located and how it is filled. This information is provided to the fuzz testing tool through its configuration file.

Messages are handled in `SConnection::processMsg()`, shown in Figure 6.2. This method is our target region for fuzz testing.



```
0000000000421B37 mov     [rbp+var_21], al
0000000000421B3A jmp     short loc_421B48

0000000000421B48
0000000000421B48 loc_421B48:
0000000000421B48 mov     rax, [rbp+sconnection]
0000000000421B4C mov     rdi, rax        ; this
0000000000421B4F call    SConnection__getFdStream
0000000000421B54 mov     esi, 1          ; nitems
0000000000421B59 mov     rdi, rax        ; this
0000000000421B5C call    FdInStream__readBuffered2
0000000000421B61 test    al, al
0000000000421B63 jnz     short loc_421B3C

0000000000421B3C
0000000000421B3C loc_421B3C:
0000000000421B3C mov     rax, [rbp+sconnection]
0000000000421B40 mov     rdi, rax        ; this
0000000000421B43 call    SConnection__processMsg

0000000000421B65 movzx   eax, [rbp+var_21]
0000000000421B69 xor     eax, 1
0000000000421B6C test    al, al
0000000000421B6E jz      short loc_421B90

0000000000421B70 mov     rax, [rbp+sconnection]
0000000000421B74 add     rax, 160h
0000000000421B7A mov     rdi, rax
0000000000421B7D call    sub_411FBC
0000000000421B82 xor     eax, 1

0000000000421C01
0000000000421C04

100.00% (599,624) (730,368) 00021B43 0000000000421B43: handle client+65
```

Figure 6.1: Region of the VNC server where messages are read and processed.



```
000000000041BD08
000000000041BD08 ; Attributes: bp-based frame
000000000041BD08
000000000041BD08 ; int __cdecl SConnection__processMsg(SConnection *this)
000000000041BD08 SConnection__processMsg proc near
000000000041BD08
000000000041BD08 this= qword ptr -18h
000000000041BD08
000000000041BD08 push    rbp
000000000041BD09 mov     rbp, rsp
000000000041BD0C push    rbx
000000000041BD0D sub     rsp, 18h
000000000041BD11 mov     [rbp+this], rdi
000000000041BD15 mov     rax, [rbp+this]
000000000041BD19 mov     eax, [rax+SConnection.state]
000000000041BD1F cmp     eax, 6
000000000041BD22 ja      loc_41BE1F      ; jumptable 000000000041BD49 default case

000000000041BD28 mov     eax, eax
000000000041BD2A lea     rdx, ds:0[rax*4]
000000000041BD32 lea     rax, off_43B5E0
000000000041BD39 mov     eax, [rdx+rax]
000000000041BD3C movsxd  rdx, eax
000000000041BD3F lea     rax, off_43B5E0
000000000041BD46 add     rax, rdx
000000000041BD49 jmp     rax                ; switch 7 cases

000000000041BE1F
000000000041BE1F loc_41BE1F:              ; jumptable 0000000000
000000000041BE1F mov     edi, 108h
000000000041BE24 call    ___cxa_allocate_exception
000000000041BE29 mov     rbx, rax
000000000041BE2C lea     rsi, aSconnectionPro ; "SConnection::p
000000000041BE33 mov     rdi, rbx
000000000041BE36 call    sub_40A290
000000000041BE3B mov     edx, 0
000000000041BE40 lea     rax, off_43A1D0
000000000041BE47 mov     rsi, rax
000000000041BE4A mov     rdi, rbx
000000000041BE4D call    ___cxa_throw

100.00% (1950,34) (591,199) 0001BD7E 000000000041BD7E: SConnection processMsg:loc 41BD7E
```

Figure 6.2: The first instructions of the `SConnection::processMsg()` function.

With this knowledge we can create the configuration file shown in Listing 6.1. The *listening* and *established* breakpoints are points that are easy to find in the target binary. The *region start* breakpoint is set to the first instruction in `SConnection::processMsg()`. Finally, the *region end* breakpoints are set at various places where we want to stop execution and return to the start.

The *region end* breakpoints include:

- normal return from `SConnection::processMsg()`,
- points where an exception is thrown,
- the point where an exception from subroutines is caught higher up in the stack,
- the point where communication with the X server is initiated.

We need to include all of these points in order to limit changes to the global state that occur within the fuzz testing region. For example, we deliberately exclude communication

```
[FuzzConfig]
pcap=vnc-1-small.pcap
logfile=vnc-1.log
dumpdir=dumps

[pinfuzz]
port=5900
;sequence=connect, r, 6, r, 9, r, 11, r, A, 13, B
sequence=connect, r, A, 6, B

bkpt_listening = 0x404eb2
bkpt_established = 0x40530c

; SConnection::processMsg
bkpt_region_start = 0x41bd08
skip_region_start = 0
bkpt_region_end = 0x41be58, 0x41bdb9, 0x41bdec, 0x421bf5, 0x406372

; rdi ... SConnection instance, FdInStream at 0x80
; in FdInStream, start of buffer is at 0x8
regionfuzz_buffer = *(*(rdi + 0x80) + 0x8)

[Packet6]
;range = 0-8
```

Listing 6.1: Configuration file for fuzz testing the VNC server with our tool.

with the X server, since this communication protocol uses sequence numbers. With our in-memory fuzz testing approach, sequence numbers would only match in the first iteration and produce errors later.

Similarly, we stop the program before a higher-level exception handler is executed, since this has unwanted side-effects that we cannot easily reverse. In particular, the server closes the client socket when an error occurs, and this would break calls to `select()` in future iterations.

6.2 Speed Comparison

Performing fuzz testing in-memory instead of opening new connections for each attempt has a potential speed advantage. Within this work, we try to compare it with different approaches. This is not an extensive benchmark, but the numbers should provide a rough idea of the performance relative to other approaches.

When testing this on Linux, a major limiting factor for TCP-based approaches is that sockets will linger in wait states for a while. We thus enable two kernel tuning options to allow reusing sockets earlier, as shown in Listing 6.2.

Setup 1. For testing, we use a simple non-forking TCP server which reads and parses messages. To establish a baseline for the comparison, we use a simple C client program opening a connections, sending a short message and closing it again. This should allow

```
sysctl net.ipv4.tcp_tw_reuse=1
sysctl net.ipv4.tcp_tw_recycle=1
```

Listing 6.2: Tuning commands for the Linux TCP stack.

us to get an idea of how much overhead we have just by repeatedly opening/closing TCP connections.

Setup 2. A simple fuzz testing tool written in Python that is in use at our institute is used to perform testing on the same binary.

Setup 3. The in-memory fuzz testing tool implemented as part of this thesis is used to test the message handling part of the binary.

Setup	Iterations	Time [s]	Rate [it/s]	Relative
Setup 1: Connection only	50 000	19.2	≈ 2 600	1.0×
Setup 2: Python fuzzing	9 172	5.0	≈ 1 830	0.7×
Setup 3: In-memory fuzzing	69 521	10.0	≈ 6 950	2.7×

Table 6.1: Measurements comparing the speed of different fuzz testing setups. All measurements were taken on Linux with a simple example server program.

Measurement results are shown in Table 6.1. We can see that the C program just opening as many connections as possible is only slightly faster than the Python fuzzer. This suggests that the TCP connection overhead occuring in the operating system is indeed a major limiting factor.

The in-memory fuzzer using Pin is doing quite well, being 2.7 times as fast as the C program opening TCP connections. For one, the TCP connection overhead is eliminated with the in-memory approach. Also, only a smaller region of the program is run in each iteration.

There are limitations to this experiment. First of all, the example program we used has very simple message parsing logic, which might potentially give some advantage to the Pin fuzzer. When the region of the program that should be fuzzed is smaller, less code needs to be executed and less memory writes need to be traced and restored.

Our benchmark also does not account for the fact that when the fuzzer actually finds bugs, it might take more time to recover from crashes and thus be slower.

CHAPTER 7

Conclusion and Outlook

In this thesis, we demonstrated that performing in-memory fuzz testing using the Pin framework is feasible. We implemented a memory snapshotting approach and explored mechanisms to deal with parallelism in target programs.

Evaluating our proof-of-concept tool, we found that applying it to real-world programs is possible with limitations. A comparison showed that performance of the in-memory fuzzer surpasses the naive network fuzz testing strategy.

The obstacles we hit lie mostly in the snapshotting mechanism. Since our tool can only capture memory contents, all other aspects of the program state leak through between iterations. In particular, I/O functions are problematic and in some cases need to be isolated first to be able to use the tool. This could be worked around by catching some system calls and handling them appropriately, but this gets very complex very quickly.

Performance-wise, more experiments could be conducted on larger programs to get more reliable results. More optimizations could be made, in particular in the implementation that records memory writes. Also, one might experiment with reducing thread synchronization overhead by disabling some parts of the synchronization protocol when dealing with single-threaded target binaries.

The tool in its current shape is a proof of concept. For real-world use, usability improvements could be made such as better error messages and more robust handling of timeouts and termination. It might also be desirable to be able to use custom scripts to modify data before it is injected into the application, to be able to handle checksums, sequence numbers and similar features of application protocols.

References

[Oeh05] Peter Oehlert. “Violating assumptions with fuzzing”. In: *Security & Privacy* 3.2 (2005), pp. 58–62.

[Wan11] Tielei Wang et al. “Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution”. In: *ACM Transactions on Information System Security* 14.2 (Sept. 2011), 15:1–15:28.

[Sut07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[Luk05] Chi-Keung Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2005, pp. 190–200.

[PinWeb] *Pin—A Dynamic Binary Instrumentation Tool.* URL: `https://software.intel.com/en-us/articles/pintool`.

[PinGuide] *Pin 2.13 User Guide.* URL: `https://software.intel.com/sites/landingpage/pintool/docs/65163/Pin/html/`.

[RealVNC] *RealVNC: VNC Open Download Page.* URL: `https://www.realvnc.com/download/open/`.