Software Fuzzing with Wireplay

Abhisek Datta
abhisek.datta@ivizsecurity.com

http://code.google.com/p/wireplay/

February 4, 2010

Contents

Abstract

This paper presents a simple yet powerful idea for rapid fuzz testing of Network Applications. Theoritically fuzzing involves supplying invalid or semi-invalid input set to the target application and monitoring for possible faults. However given the wide set of different protocols publicly available along with proprietory protocols with non-public specifications, writing fuzzers for each the different protocol although more complete but highly time consuming. Wireplay can be used as a quick approach to preliminary fuzzing of applications implementing totally unknown/custom protocol. The fundamental concept of Wireplay is to read pcap dumps of valid communication between our target server and its original client application, modify the original client-to-server data to introduce possible faults in the server and replay it to the server. Wireplay uses stream socket to communicate with server and uses only the TCP Payload part from the pcap dumps hence it avoids any of the internal details of handling TCP keeping itself to minimal and simple.

1 Introduction

This paper provides an introduction to Wireplay, its development goals, design and how to use Wireplay to conduct various Software Testing exercise including Fuzz Testing. It will also provide a brief introduction to the theory of Fuzzing or Fuzz Testing for readers new in this domain.

It is assumed that the readers will consider this paper to be a practical guide to using Wireplay along with enough information to customize and/or enhance Wireplay. This paper does not provide any research result conducted by the author.

Wireplay is available free for download along with Source Code at:

`http://code.google.com/p/wireplay/`

1.1 Fuzzing

Theoritically fuzzing involves sending invalid/semi-invalid data set to a target application and monitoring for possible fault. Network Servers are one of the most common server subjected to fuzzing with the intention of finding bugs with Security Impact.

1.1.1 Blind Fuzzing

An application can be fuzzed by feeding completely invalid or random input data to it without any respect for its expected protocol of communication. This kind of fuzzing which involves feeding only random unstructured data to a target application is called Blind Fuzzing.

It might be possible to send arbitrary and/or completely invalid data to a target application with the intention of triggering a possible bug in the target application, however if the input data is completely invalid, then the coverage of testing will be very low ie. the application most probably will detect the invalid nature of its input and hence will not be executing its various code paths.

Considering a HTTP Server which normally accepts a request like:

```
GET / HTTP/1.1
Host: target.com
User-Agent: TestClient
```

If however during a fuzzing session, a completely invalid request in the form of say `AAAAAA....AA` is sent to the HTTP Server, although it might trigger a possible bug but the chances are very low because HTTP Servers are supposed to process HTTP Verbs like GET/PUT/HEAD etc. and when it sees an invalid HTTP Verb in the input, the chances of executing maximum code path is very low hence reducing the overall testing coverage. Similarly if a possible bug exists in the target HTTP Server in parsing of say the `User-Agent` header, then completely arbitrary or blind fuzzing will never hit the buggy code.

1.1.2 Block Based Fuzzing

Block Based Fuzzing can be considered as a theoritical model of the process of fuzzing where the original valid input to an application is reduced/broken into blocks of related or unrelated data and each block is periodically altered while mostly keeping the other blocks to its original value. This is a very comprehensive method of fuzzing a target application with considerably high coverage. This technique of fuzzing is already well described [1].

Considering an HTTP request again of the form:

```
GET / HTTP/1.1
Host: target.com
User-Agent: TestClient
```

The request string here can be broken down or tokenized into a set of tokens or blocks consisting of say `GET`, `/`, `HTTP`, `Host`, `target.com`, `User-Agent` etc. and each of the tokens are fuzzed periodically. Following this technique a single fuzz string can then look like the following string considering the token `target.com` is fuzzed:

```
GET / HTTP/1.1
Host: AAAAAAAAAAAAAAAAAAAAAAAAAA....AAAAAA
User-Agent: TestClient
```

2 Design Goals

Wireplay is designed with the goals of implementing a minimalist approach to replay pcap [2] dumped TCP sessions with modifications as required. The aim of the Wireplay Project is to build an usable but simplistic tool which can help in replaying selected TCP sessions. It can play both client as well as the server during a replay session.

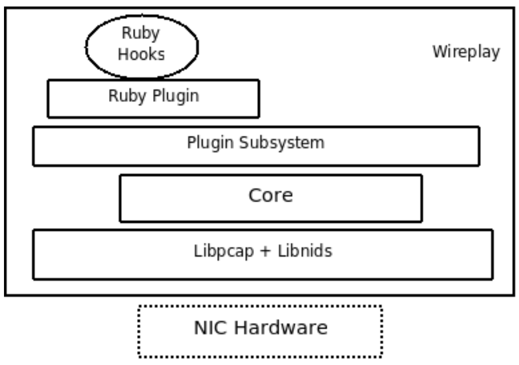
Replay Attacks however doesn't work against protocols which are cryptographically hardened or implements protocol specific replay prevention mechanism like challenge/response etc. Wireplay implements a plugin/hook subsystem mainly for the purpose of working around those replay prevention mechanism and also perform a certain degree of fuzz testing.

It is important to mention here that The TCPReplay Project [3] also develops a set of tools for replaying pcap dumped packets at various layers of the network stack. The primary difference of Wireplay with TCPReplay is that Wireplay works only for TCP Sessions at a stream socket level unlike TCPReplay which works on a per packet replay basis. Wireplay is capable of reconstructing a TCP Session from a pcap file using the libnids [4] library.

3 Architecture

3.1 Software Architecture

The basic software architecture of Wireplay can be described by the following diagram:



3.2 The Hook Subsystem

The hook/plugin subsystem is designed to allow external/non-core code to manipulate a replay session ie. provide APIs to intercept and alter each packet during a live replay session.

Hooks are called by the core during occurrence of certain events. Wireplay defines the following events for hook invocation and during each invocation the corresponding method/function handling the event is called for each registered hooks:

- `START`: This event is triggered when a new connection is established with the target server.
- `STOP`: This event is triggered when a connection with the target server is terminated/closed.
- `DATA`: This event is triggered when data is sent or received from the target server.
- `ERROR`: This event is triggered on occurrence of an error which is identified by an integer error code.

3.2.1 An Example Hook

In order to write a hook, the first requirement is to define a hook descriptor and use the `w_register_hook(...)` API to register the hook with `Wireplay` core:

```
static
struct w_hook my_dummy_hook = {
   .name = "dummy",
   .init = dummy_init,      /* called when the hook is initialized */
   .start = dummy_start,    /* called when a TCP session starts */
   .data = dummy_data,      /* called when socket IO is performed */
   .stop = dummy_stop,      /* called when TCP session is closed */
   .error = dummy_error,    /* called to notify errors */
   .deinit = dummy_deinit   /* called when wireplay exits */
};

static
struct w_hook_conf my_dummy_hook_conf = {
   .search_path = NULL,     /* unused currently */
   .ext = NULL,             /* unused currently */
};

/* called when the plugin is to be loaded */
int w_dummyhook_init()
{
   cmsg("Initializing dummy hook");
   w_register_hook(&my_dummy_hook, &my_dummy_hook_conf);
}

/* called when the plugin is to be unloaded */
void w_dummyhook_deinit()
{
   cmsg("Deinitializing dummy hook");
}
```

Once the hook is registered, the `Wireplay` core will automatically call appropriate callback functions in the hook (specified by the hook descriptor) on occurrence of specific events:

```
static
void dummy_start(struct w_hook_desc *w)
{
   cmsg("START event occurred");
}

static
void dummy_data(struct w_hook_desc *w,
               uint8_t direction,
               char **data,
               size_t *len)
{
   cmsg("DATA event occurred");
}

static
void dummy_stop(struct w_hook_desc *w)
{
   cmsg("STOP event occurred");
}

static
void dummy_error(struct w_hook_desc *w, int error)
{
   cmsg("ERROR event occurred");
}
```

Readers are suggested to look into sample hooks in `hooks/` subdirectory from the `Wireplay` source to have a better idea about writing real life hooks for practical purposes.

3.2.2 The Ruby Hook API

The Ruby Hook is designed as a bridge between `C` and `Ruby` and to enable `Wireplay` users write packet manipulation hooks in `Ruby` language.

A `Wireplay` hook can be written in Ruby language following the same model as described above.

```
class MySampleHook
   def on_start(w_desc)
   end

   def on_stop(w_desc)
   end

   def on_error(w_desc, error)
   end

   def on_data(w_desc, direction, data)
   end
end

Wireplay::Hooks.register(MySampleHook.new())
```

A simple way to fuzz a server is to alter the original client to server data when the `DATA` event is triggered and the corresponding callback method is called:

```
def on_data(w_desc, direction, data)
   return nil unless (direction == Wireplay::REPLAY_CLIENT_TO_SERVER)

   offset = rand(data.size)
   return (data[0, offset] +
            ("A" * 65535) +
            data[offset, data.size - offset])
end
```

4 Using Wireplay

`Wireplay` is a command line tool tested only on GNU/Linux platform so far which uses libraries like libnids, libpcap, libruby etc. A basic command line use of `Wireplay` can be described as:

```
./wireplay -K --role client --port 80 \
   --target 127.0.0.1 -L -F ./pcap/http.dump
```

The above runs wireplay with TCP checksum calculation disabled, replaying an HTTP session from ./pcap/http.dump file.

4.1 Compilation

`Wireplay` has the following platform dependencies:

- Libpcap [2]
- Libnids [4]
- Ruby Development Packages [5]

`Wireplay` uses a patched version of libnids library so it is suggested to link `Wireplay` with the version of libnids supplied with its source. For a detailed guide on compilation of `Wireplay`, readers are suggested to refer:

`http://code.google.com/p/wireplay/wiki/WireplayCompile`

4.2 A Basic Run

`Wireplay` acts as a replay agent reading TCP session from a valid pcap dump and replaying the TCP payload read from the packets to the target TCP endpoint over TCP Stream sockets. Thus in order to use `Wireplay`, the first requirement after installation is to obtain a TCP session dumped in a file in pcap format. This can be achieved easily using tools like Wireshark [6]. Once a pcap dumped TCP session is obtained, it can be replayed using `Wireplay` easily:

```
test@test:~/ROOT/codes/wireplay$ ./wireplay -r client -K \
> -t 127.0.0.1 -p 80 -F ./pcap/http.dump
[*] Loading TCP sessions from pcap dump.. count:1
SHOST         SPORT     DHOST            DPORT    CDSEQ         SDSEQ
172.16.6.16   57895     208.109.181.107  80       0x3e22b8f0    0xcb8c0803

Enter session no. to replay: 1
[*] Initializing hooks
[*] Initializing Ruby hook (Hook File: (null))
[*] Registering hook: ruby
[*] Connecting to target host..
[*] Disabling NIDS checksum calculation
[*] Session start event raised
[*] Run Count: 1 Server data: 600 Client data: 1339
[*] Session stop event raised
```

References

[1] *The Advantages of Block-Based Protocol Analysis for Security Testing*, Dave Aitel, Immunity Inc, 2004.

[2] *Libpcap*, http://www.tcpdump.org/

[3] *TCP Replay*, http://tcpreplay.synfin.net/trac/

[4] *Libnids*, http://libnids.sf.net

[5] *The Ruby Language*, http://www.ruby-lang.org

[6] *Wireshark*, http://www.wireshark.org