Writing a fuzzer using the Fuzzled framework

Tim Brown

February 24, 2008

<mailto:timb@nth-dimension.org.uk>
<http://www.nth-dimension.org.uk> / <http://www.machine.org.uk>

Abstract

Fuzzled is a powerful fuzzing framework. Fuzzled includes helper functions, namespaces, factories which allow a wide variety of fuzzing tools to be developed. Fuzzled comes with several example protocols and drivers for them. The aim of this paper is to document a practical process for using the framework to find new vulnerabilities.

1 Technical Details

1. Protocol fundamentals

   The first task in developing a fuzzer using the Fuzzled framework is to get a fundamental understanding for the protocol you're attempting to fuzz. It's fair to say that there are a number of sources that can prove beneficial in this regard. These include:

   (a) Documentation

   Nothing beats an RFC in this regard, especially since they often include BNF specifications for how the protocol is intended to be implemented. For example, this is a snippet from the RFC for IRC[1].

```
<message> ::= [':' <prefix> <SPACE> ] <command> <params> <crlf>
<prefix> ::= <servername> | <nick> [ '!' <user> ] [ '@' <host> ]
<command> ::= <letter> { <letter> } | <number> <number> <number>
<SPACE> ::= ' ' { ' ' }
<params> ::= <SPACE> [ ':' <trailing> | <middle> <params> ]
<middle> ::= <Any *non-empty* sequence of octets not including
SPACE or NUL or CR or LF, the first of which may not be ':'>
<trailing> ::= <Any, possibly *empty*, sequence of octets not
including NUL or CR or LF>
<crlf> ::= CR LF
```

   As you can see, the BNF specification define a message as been a sequence of subblocks. We could use this later on when we're defining our base cases. Just remember though that these only define how the protocol is meant to behave and we're looking primarily for deviations since these are a good source of vulnerabilities.

   Of course, RFCs aren't the only source of documentation about a protocol. You might also find documentation in SDKs or technical literature on the vendors web site. Hell, there may even have been other researchers who have published work on the protocol. In any event, documentation is definately the first place to look when you're trying to understand a new protocol.

(b) Observation

So after devouring any documentation available, is it time to go find some vulnerabilities? For me, the next step is to observe the target. I'll go grab a copy of the application and fire it up on a test system. Sometimes I'll use a virtual machine and sometimes, I'll need the real thing. It's useful to have a good selection of hardware and OS to choose from here since the really interesting stuff often only runs on the more obscure platforms, but VMware and a raft of virtual machines is a good starting point.

Anyway, once I have it up and running I'll monitor it in typical usage, I'll make use of the vendor supplied client tools as well as tools for observation such as packet sniffers, debuggers, function tracers and file monitors. You can't really find vulnerabilities unless you have some idea about the expected behaviour of the application after all, especially if the application extends an existing protocol in a proprietry manner.

Of particular interest here is the cause and effect of event you observed. How does super-whizzy-HR-application client authenticate when you enter your password? Does it use repeatable byte sequences to query its server component? Are any characters encoded before sending?

(c) Static analysis

Finally, I'll normally try a little static analysis of the binaries based on what I'm gleaned so far. I'm a lousy excuse for a reverse engineer, so I normally limit myself to tools that process the binaries in some manner. What libraries does the application use? Does it use any functions that might lead to security flaws, can I find the command strings embedded in the binary? Are there any strings I haven't seen yet? Are there any odd strings? For example, I initially identified the existence of a heap overflow in Visionsoft Audit[2] by using the strings tool to locate commands I'd already observed on the wire and in so doing found the hidden (and vulnerable) `LOG` command.

2. Building a base request

Once you've got an idea how the protocol works, it's time to generate some base requests. In the most simple case this can be a simple string, but for more complex cases (particularly where the protocol is in binary), it is often find it easier read it in directly from a previous packet capture. You might even choose not to start with a base request at all, and simply define the entire request using factories. As an aside, `Fuzzled::Protocol::HTTPInject` does the former (reads from WebScarab logs), whilst `Fuzzled::Protocol::SOCKS4` does the latter (defines the full request using factories).

3. Using namespaces

You also need to figure out which elements of the base requests you wish to fuzz. Essentially, you need to consider how the developer is likely to parse your base request and then out guess them. Obviously not all classes of vulnerability are applicable to any given application so you'll also need to consider how the application has been developed and what technologies it makes use of. Some obvious things to consider:

- Buffer overflows
- Integer overflows
- Format strings
- Parsing bugs
- Timing attacks
- Information disclosure
- Null pointers
- SQL injection
- Javascript injection

- LDAP injection

Fuzzled has namespaces geared to help you identify all of the above.

Essentially, Fuzzled's namespaces come with a `generate()` function which returns an array of strings that relate to a given namespace. Typically, you should then use the `Fuzzled::Factory::BruteForce` factory to iterate through the array, using each string in turn to construct futher more complex strings with which to fuzz the application.

4. Introducing Forker

   Fuzzled 2.0 marks the introduction of `Fuzzled::Helper::Forker` base class from which all protocols are derived. Drivers call the `manager()` derived from the Forker base class to start a fuzzing session. This in turn calls `produce()` and `consume()` from the fuzzer concerned to handle the actual production and consumption of request strings. In the case of the `Fuzzled::Protocol::HTTPFormsAuthenticate->produce()` function for example, it calls `queue()` from the Forker base class to place these request strings in to the shared memory pools.

5. Putting it all together with factories

   Fundamentally, a protocol fuzzer written using the Fuzzled framework is simply a set of nested loops, which iterate through generating each of the test cases for a given factory. So lets look at a real world example of how this works:

```
$requeststring = "HEAD / HTTP/1.0\n";
$requeststring = $requeststring . "Host: " . $self->{'hostname'} . ":" .
$self->{'portnumber'} . "\n";
$requeststring = $requeststring . "User-Agent: Mozilla/5.0 (compatible; HTTP)\n";
$requeststring = $requeststring . "\n";
$overflowhandle = Fuzzled::Factory::Repeat->new("A", 0, 4096);
while ($overflowstring = $overflowhandle->generate()) {
$fullrequesthandle = Fuzzled::Factory::Replace->new($requeststring, [["HEAD", ""],
        [" /", " /"], ["Host: " . $self->{'hostname'} . ":" .
        $self->{'portnumber'}, "Host: "], ["User-Agent: Mozilla/5.0 \\\(compatible;
        HTTP\\\)", ""], ["Mozilla/5.0 \\\(compatible; HTTP\\\)", ""]], $overflowstring);
while ($fullrequeststring = $fullrequesthandle->generate()) {
$self->queue($fullrequeststring);
}
}
```

   (a) The base request

   Here, we're looking a section of code designed to fuzz the `HEAD` method for potential overflows. Since this is a fairly trivial request, I've defined base request in the code inline:

```
"HEAD / HTTP/1.0\n"
"Host: " . $self->{'hostname'} . ":" . $self->{'portnumber'} . "\n";
"User-Agent: Mozilla/5.0 (compatible; HTTP)\n";
"\n";
```

   (b) Creating the full request

   First we create an instance of the `Fuzzled::Factory::Repeat` factory. This factory will repeat the initial string in ever decrementing quantities. In this case, the first call to `generate()` will return the string `"A"x4096` whilst the last call will return an empty

string. Next we create an instance of a second factory `Fuzzled::Factory::Replace`. This factory essentially takes an initial string, an array of target strings and a string to replace the targets with. Each call to `generate()` here, will lead to a different target string being replaced,the first call resulting in `"HEAD"` being replaced whilst the last call will result in `"Mozilla/5.0 (compatible; HTTP)"` being replaced.

There are also number of other factories available including:

- `Fuzzled::Factory::BruteForce`
- `Fuzzled::Factory::Increment`
- `Fuzzled::Factory::Pattern`
- `Fuzzled::Factory::Date`
- `Fuzzled::Factory::Pad`

Which allow various manipulations of the request, all of which should be self explanatory and of course there is nothing to stop you writing your own.

(c) Threading

Most of the fuzzers I write with the Fuzzled framework are multi-threaded, consisting of a producer thread and multiple consumer threads. This makes use of `Fuzzled::Helper::SharedMemory` to allow the various threads to communicate with each other, although in Fuzzled 2.0, the use of the `Fuzzled::Helper::Forker` abstracts this process for us. There's no particular reason for this and indeed there may be occasions where you want to write single threaded fuzzers. Fuzzled does nothing to prevent this.

6. Fuzzled tricks of the trade

(a) `Fuzzled::Factory::Pattern`

The `Fuzzled::Factory::Pattern` factory was inspired by `Rex::Text.pattern_create` in Metasploit. It allows patterns of arbitrary length to be created whilst avoiding so called bad characters. The aim of the factory is to speed up the process of identifying offsets, particularly in the creating of buffer overflow style attacks. It does this by seeding the random number generator based on the length of string required and then generating a string within the constraints specified. It can be a useful alternative to `Fuzzled::Factory::Repeat` although obviously you need to be aware of any special characters which might inhibit its use. The obvious case of a bad character is `\x00` but there are doubtless others, depending on the protocol being fuzzed.

(b) `Fuzzled::PacketParse::Separate`

Packet parsers are designed to break packets down into more meaningful components. `Fuzzled::PacketParse::Separate` is the first such packet parser. It is designed to break down packets into more meaningful components based on commonly used separators. By breaking packets down like this, particularly where binary protocols are in used, it may become easier to see the structure of the packets exposing the types and lengths of elements with the packet. This should allow more meaningful fuzzing to occur.

7. Using `Fuzzled::Protocol::Generic`

Another new feature in Fuzzled 2.0 is `Fuzzled::Protocol::Generic`. This protocol doesn't in itself target a specific service but rather allows the definition on further protocols using an XML schema. This introduces Fuzzled's first dependency, on XML::Simple, but I think you'll find it's worth it.

Let's start by looking at an example protocol definition marked up using `Fuzzled::Protocol::Generic`'s schema.

```
<fuzzled start="user">
```

```
        <send name="user">
                <request marker="{FUZZ}" default="anonymous" prefix="" postfix="">
                        <startstring>USER {FUZZ}\r\n</startstring>
                </request>
                <check type="buffer"/>
                <success goto="password">
                        <pattern>^(331)(.*)</pattern>
                </success>
        </send>
        <send name="password">
                <request marker="{FUZZ}" default="test@example.com" prefix=""
                postfix="">
                        <startstring>PASS {FUZZ}\r\n</startstring>
                </request>
                <check type="buffer"/>
                <success>
                        <pattern>^(530)(.*)</pattern>
                </success>
        </send>
</fuzzled>
```

Line 1 opens the definition and specifies which `send` requests should be considered starting points for Fuzzled. In this case, there is just one, `user`, but other protocol definitions may list further starting points, separated by commas. Fuzzled will use each starting point in turn as a basis for a simple state machine. Looking at the `user` send request, you will see that it defines a request, a check and a success. Lets look at them in turn. Line 3 starts a request. We give a `default` value, a `marker` which indicates where our malicious data should be placed, as well as `prefix`es and `postfix`es for the malicious data. Line 4 defines the request to be sent. Nothing will be read by the protocol until a full manipulated request has been sent. Line 6 specifies the kind of `check` we wish to carry out. For now only `buffer` is supported but this will likely change in future releases. For the record this causes the protocol to replace the `marker` with each of `"A"x4096`, `""` and the `default`. Finally, lines 7 and 8 handle `success` cases. The use of `goto` indicates that in the event that `pattern` is matched in response to the request that a second `send` request named `password` should then be sent, without tearing down the connection. When a `goto` is triggered, the protocol will move straight to the next `send`request. It will only return if a `success` case fails or if all further child `send` requests are completed.

As a comparison, let's look at another defintion:

```
<fuzzled start="get, hostname">
        <send name="get">
                <request marker="{FUZZ}" default="" prefix="/" postfix="">
                        <startstring>GET {FUZZ} HTTP/1.0\r\nConnection:
                        Keep-Alive\r\n\r\n</startstring>
                </request>
                <check type="buffer"/>
                <success goto="hostname">
                        <pattern>^(HTTP/1.1) ([\d]{3})(.*)</pattern>
                </success>
        </send>
        <send name="hostname">
                <request marker="{FUZZ}" default="{HOSTNAME}:{PORTNUMBER}"
```

```
                prefix="" postfix="">
                        <startstring>GET / HTTP/1.1\r\nHostname: {FUZZ}\r\n
                        Connection: Keep-Alive\r\n\r\n</startstring>
                </request>
                <check type="buffer"/>
                <success>
                        <pattern>^(HTTP/1.1) ([\d]{3})(.*)</pattern>
                </success>
        </send>
</fuzzled>
```

Note the difference on line 1. This protocol definition has two `send` request that should be considered starting points for fuzzing. The protocol will send `get`, `get` and `hostname` and `hostname` send requests depending on the response it recieves from each `check`:

Start of `get send` requests:

```
GET / HTTP/1.0
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: 127.0.0.1:80
Connection: Keep-Alive
GET / HTTP/1.1
Hostname:
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: AAAAAAAAAA
Connection: Keep-Alive
GET / HTTP/1.0
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: 127.0.0.1:80
Connection: Keep-Alive
GET / HTTP/1.1
Hostname:
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: AAAAAAAAAA
Connection: Keep-Alive
GET /AAAAAAAAAA HTTP/1.0
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: 127.0.0.1:80
Connection: Keep-Alive
GET / HTTP/1.1
Hostname:
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: AAAAAAAAAA
Connection: Keep-Alive
```

Start of `hostname send` requests:

```
GET / HTTP/1.1
Hostname: 127.0.0.1:80
Connection: Keep-Alive
GET / HTTP/1.1
Hostname:
Connection: Keep-Alive
GET / HTTP/1.1
Hostname: AAAAAAAAAA
Connection: Keep-Alive
```

We could disable the inclusion of the child `hostname` send requests by `get` by removing the goto directive on line 7.

8. Monitoring for vulnerabilities

   Ah yes, vulnerabilities, that's why you're playing with Fuzzled after all. I tend to make use of a number of techniques to monitor for vulnerabilities thrown up by the fuzzers.

   (a) Inline code
   Sometimes the easist way to spot a vulnerability is simply to include inline code to parse the responses. This seems to work best for cases where there is an expected behaviour. For example, I used this approach in `Fuzzled::Protocol::HTTPDirBuster` where I was expecting a 404 response code to be returned by the web server for each of my requests. Using a regular expression, I was quickly able to confirm that this was the case for each of my requests.

   (b) Logging the input/output
   This is better where the expected response from the application is undefined, but where you are assured of getting a response one way or another. Fuzzled includes a logging function within `Fuzzled::Helper` which `Fuzzled::Protocol::HTTPInject` makes use of. Essentially, for this fuzzer (which fuzzes based on WebScarab logs), my injections (the modified parts of my full requests) include a marker and both the input and output is logged. At the end of a run, I can grep the logs to see how the application behaved in response to my injections.

   (c) Monitoring the application
   This essentially takes the same form as observation during the Protocol fundamentals stage of writing a fuzzer, since you're now actively probing the application in a non standard manner, you may well learn more about how the application and the protocol it implements responds to unexpected input. For example, I'll run a packet sniffer when I fuzz binary protocols since they give me a chance to review responses in another form which may mean I spot something I would have otherwise missed.

2 Changes

0.0.0-1.1.0 Updated with details of Fuzzled 2.0
0.0.0-1.0.0 First published

References

[1] *http://www.faqs.org/rfcs/rfc1459.html*

[2] *http://www.securityfocus.com/bid/25153/*