Fuzzit: A Mobile Fuzzing Tool

Kevin Mahaffey
John Hering
Anthony Lineberry

Flexilis – http://www.flexilis.com

Introduction to Fuzzit

Fuzzit is a tool designed to find vulnerabilities in mobile devices. It implements a testing technique that most readers will be familiar with called fuzzing. In short, fuzzing aims to trigger software faults by automatically-generating and injecting unusual, unexpected, or out-of-specification input into a system. Such input typically includes malformed data structures/formats in both expected and unexpected system states. In a fuzzing system, there are four major components that contribute its overall effectiveness: a generation system, a state management system, an injection system, and a result analysis system. As we will discuss below, Fuzzit was written from the ground up to address the problems faced when auditing mobile devices and mobile-specific protocols.

Why did we need to write something new?

Although currently available fuzzers are fine for most uses, we believe that mobile is a unique enough problem domain that existing tools would require significant changes to address the problems we face in fuzzing mobile devices. We also wanted to share a few fuzzing techniques that we've found to be successful over the years.

Fuzzit is built to address the problems with fuzzing mobile devices.

Most importantly, the attack surface of a mobile device has significantly more breadth than that of a typical PC or server. We cannot solely focus on protocols that are borne over the TCP/IP, as mobile devices implement additional protocol stacks such as Bluetooth, WAP, and NFC. A mobile fuzzing system must be flexible enough to support not only the generation of data for various protocols and file formats, but also the state management and injection requirements for those protocols and files.

Fuzzit is designed with a generic core that performs functionality common to various protocols and file formats. The framework is designed to prefer configuration over customization. For example, when generating integers as part of a file type or format, Fuzzit uses an `IntElement`. By default, an instance of `IntElement` generates a 1-byte unsigned integer. If a 3 byte little-endian integer was needed to fuzz a given protocol, instead of building a new element (e.g. `LE32IntElement`) you can simply configure an `IntElement` to act as desired by sending `has_length(3)` and `has_endianness(:little)` to it. To minimize configuration, the framework uses intelligent defaults wherever possible. The default `IntElement` (1 byte) will generate fuzz values that match the configuration of that element ("\x00", "\xff", "\x7f", "\x80", etc.). The 3-byte little-endian integer element by default will generate fuzz values that match its configuration ("\x00\x00\x00", "\xff\xff\xff", "\xff\xff\x7f", "\x00\x00\x80", etc.).

Fuzzing wireless protocol stacks such as Bluetooth, WAP, NFC, and Wi-Fi is usually best performed in communication with physical hardware, rather than a simulated or emulated environment. This means that Fuzzit must be able to interface with a radio to communicate with the device under test. The Fuzzit generation system makes no assumptions about radio-specific injection, state management, or result analysis systems. The generation system can simply be asked to create PDUs (protocol data units) that

use supplied session or connection context information (e.g. dynamic channel numbers, sequence identifiers, port numbers). Furthermore, the generation system is built to generate and parse "good" PDUs for use in setup, teardown, and state management from the same data structure definitions used to generate fuzzed PDUs. More on the re-use of data structure definitions is discussed below.

When fuzzing wireless protocols on mobile devices, we often have no access or limited access to the device under test. In order to adequately categorize the results from a fuzzer's test case, we must either be able analyze data returned from a device or perform additional checks on the device (e.g. ping, current state determination) to decide if any sort of fault occurred. Needless to say, this process is a lot more complicated than attaching a debugger to a local process. Fuzzit is designed to allow easy implementation of arbitrary result analysis systems without requiring modification to the fuzzer's other systems. This modularity promotes the building generic result analysis systems that can be used to fuzz many different types of software on a given platform or many different types of platforms implementing a given protocol (e.g. a gdbserver-based result analysis system for Android devices, a Bluetooth disconnection reason analysis tool for use on multiple platforms).

To fuzz protocols and file formats that cannot be accessed remotely or are difficult to adequately access remotely, we need programmatic access to a device. Specifically, we need an agent on the device that can implement injection and result reporting. Because no single agent can run on all mobile platforms, we need multiple agents with a common interface to the fuzzer's host so that we can maximally re-use code between platforms. Fuzzit supplies an agent communication protocol that allows a host to send test cases against one or more physical or emulated devices simultaneously.

Each mobile platform tends to have peculiarities that require customizations for a fuzzing system to adequately target it. For example, analyzing Bluetooth disconnection event reasons (e.g. baseband timeout, remote radio explicitly closed connection) that reliably identify software faults on one platform often provides no insight on another. In developing earlier fuzzers that have targeted mobile devices, we've had to change substantial amounts of code to accommodate for peculiarities of one system or another because of assumptions that were baked in from the start. Fuzzit is written to be loosely coupled so that we can isolate changes required for a specific platform to only those components that are affected by the peculiarities. That way, the core components do not assume behavior of a specific device and remain unchanged.

Perhaps most relevant to someone performing a security audit, mobile protocols are evil. Under the banner of maximizing efficiency, robustness, and flexibility, such protocols liberally add complexity. Mobile protocols often use complex data structures, curious encoding methods, and well-arrowed state graphs. In order to be able to create a fuzzer in a timely manner, a generation system for mobile protocols must be VERY flexible and allow a large amount of customization on a per-protocol basis without significant amounts of new code. Fuzzit is written to allow pre-defined functionality to be extended dynamically without having to write new core elements. For example, if a protocol serialized an arbitrary length integer by reserving the most significant bit of each byte as a continuation field, `IntElement` could generate this field by being sent the following:

```
encodes_with do |instance, value, context|
  out = ""
  i = 0
  while (value>>(7*i)) > 0 do
    out<< ((value>>(7*i) & 0x7f) | (i == 0 ? 0 : 0x80))
    i += 1
  end
  out.reverse[1]
end
```

Mobile protocols also make frequent use of non-octet aligned fields, requiring the generation system to be able to fuzz bitwise elements. Fuzzit implements bitwise support by allowing elements to declare that they represent an arbitrary number of bits. Serialization to a byte stream is taken care of by a `BitBlock` which can be extended to support arbitrary bit/byte ordering conventions.

It is important to note that having a strong generation system can tempt the author of a fuzzer to err on the side of implementing a given protocol rather than building a tool to break it. Fuzzit cannot supplant creative insight into how to break software. When implementing a fuzzer with Fuzzit, one must take care to include fuzzed values that break the structure of a protocol in addition to fuzzing invalid values in a valid structure. For example, in the above example of an arbitrarily length integer field, the exemplary encoder does not allow the creation of a field with no final byte. Because Fuzzit allows custom encoders as well as custom fuzzed values, a value for a particular element may be defined so as to include arbitrary encoder parameters. An example value might be `[0xdecaf, :no_terminator]` with the encoder defined so as to respond to the :no_terminator parameter appropriately.

What else is new?

Besides addressing the major issues we've found with fuzzing mobile devices, Fuzzit implements a few other fuzzing concepts that we've found to be successful over the years. Even though Fuzzit has been built for mobile, we believe it to be an excellent general purpose fuzzer as well.

In fuzzing, and testing in general, there is a tradeoff between time required to run a suite of tests and the depth of conditions tested. The depth of conditions has two degrees of freedom. First, the author of a fuzzer may choose how many different variations a given element (e.g. integer, string, length field) has. The number of variations is fairly straightforward, as most fuzzers simply define a corpus of known integers, strings, etc. that are likely to cause errors. Second, the fuzzing framework must choose which variation of each element in a data structure to test in each test case. One approach is to build a set of test cases by specifying a known good PDU or file that fuzzes one element at a time in each test case. In the case of fuzzing an HTTP GET request, this could mean taking a valid request and manipulating a single delimiter or string at a time in each test case. This approach has the advantage of a small number of test cases for each test definition, but is likely not to find bugs that rely on two or more invalid inputs. Another approach is to use a Cartesian product, where a test case for all possible combinations of element variations is generated. This approach has the advantage of covering a wide range of the input space, but, when fuzzing a complex data structure that has many elements needing to be fuzzed, the

[1] Ruby automatically returns the result of the last evaluation from a method, so an explicit `return` is unnecessary.

number of test cases is likely to explode. Fuzzit takes a hybrid of these two approaches by iteratively testing an increasing number elements being fuzzed at a time, with all other elements taking expected (i.e. good) values. It first tests one element being fuzzed at a time, then two, three, etc. The assertion here is that there are diminishing returns as more elements fuzzed simultaneously. This combinatorial scheme has an advantage in that it allows a researcher to decide how deeply he or she wishes to fuzz, with the test cases most likely to cause errors being run first.

Besides fuzzing data content, we've found that fuzzing state transitions is a valuable part of any security audit. In our experience, simply injecting well-formed data at unexpected times has yielded some very *interesting* software flaws. Using both well-formed and malformed data at various states is even better. Fuzzit includes a core state management system that can be extended for specific test scenarios.

For a protocol that is delivered on top of another in a stack (e.g. HTTP borne on either TLS or TCP), fuzzing multiple protocols in the stack as opposed to just the top protocol has turned up some interesting errors for us in the past. Although this would certainly cause combinatorial explosion for many fuzzers, Fuzzit's approach to combinatorics ensures that fuzzing a full stack of protocols simultaneously in moderate depth is feasible in a reasonable timeframe. As a note, fuzzing a stack of protocols is usually only effective if the target implementation is leaky, that is, if the top protocol is not well encapsulated from lower layer protocols. For example, TCP and UDP are typically very well encapsulated via a socket interface, so trying to target the TCP reassembly subsystem while fuzzing an HTTP server is unlikely to find faults (I would love to be proven wrong on this!). Bluetooth and WAP, however, tend to be quite leaky, as lower protocols on the stack are often not well encapsulated.

When working with stateful protocols or multiple test configurations for a given protocol or file format, having to re-define the same data structure or PDU multiple times is not only tedious, but also contributes to brittle code. Fuzzit is built to use the same data definition format for parsing valid data, generating valid data, and generating fuzzed data. The data parsing, fuzzing, and generation methods of Elements and Blocks (the two primitive types in Fuzzit) are stateless, meaning that all information needed to parse, generate, or fuzz is passed into the methods and not stored in the Element or Block Objects. Statefulness is pushed as far towards the edges of the system as possible. For example, an injection system and state management system must be stateful to keep track of the position in a test suite or the position in a protocol's state graph, respectively. None of the core objects such systems rely on are stateful.

How do I use it?

As a note, you'll notice that we spend a lot of thought on software and API design so that Fuzzit is as flexible as possible. That flexibility not only makes writing new fuzzers for never-before-thought-of scenarios easy, but also allows us to continuously improve the architecture of the framework. From the time of writing to when you read this, some APIs may have changed, so be sure to check out the Fuzzit page at http://www.fuzzit.com to get the latest documentation.

Fuzzit is implemented in Ruby, a dynamic object-oriented programming language. Key considerations for choosing Ruby include its ability to implement Domain Specific Languages and support for functional programming features—plus it is fun to write!

The most basic part of Fuzzit is the humble `Element`. `Elements` represent a field in a protocol or file format that can take a value. Different types of `Elements` (e.g. `IntElement`, `CrcElement`, `StringElement`) are implemented as subclasses of `Element`. An `Element` is responsible for converting between serialized data and symbolic representations of data.

A discussion of `Element` is incomplete without bringing in `Block`. A `Block` can hold multiple `Elements` and even other `Blocks`.

A new `Block` can be instantiated with several `Elements` inside of it as follows:

```
block = Fuzzit::Block.new do |b|
  b.IntElement(:elem1)
  b.IntElement(:elem2)
end[2]
```

An `Element` is always declared with a name, while a `Block` is optionally declared with a name. When a `Block` is declared without a name, all symbols used to generate, parse, or fuzz a data structure are merged with the parent. If a `Block` is declared with a name, that name is used to create a separate naming layer. This way a `Block` can be used to encapsulate a protocol borne on another or simply to group together related elements for use with by a `LengthElement` or other form of reference.

An `Element` may be sent messages to customize its behavior:

```
b.IntElement(:elem1).has_endianness(:little).has_length(13,:bytes)
```

One or more nominal values for an `Element` may be defined so that when fuzzing a different `Element`, this `Element` will cycle through its nominal values.

```
b.Element(:elem1).has_value("hello")
b.Element(:elem2.has_values(:type_1 => "1", :type_2 => "2", :type_3 => "3")
```

An `Element` may also have values that are dynamically evaluated. One of the values an `Element` takes may be a ruby `Proc` object (e.g. `lambda`) that accepts 2 parameters, the instance of the `Element` and the context of the current data structure being written.

```
b.IntElement(:random).has_length(1,:byte).has_value( lambda{|inst, ctxt| rand(0x100) } )
```

---

[2] Rubyists will notice that the above syntax is different from the standard method of instantiating a class. `b.IntElement(:elem1)` is effectively `b.add(Fuzzit::IntElement.new(:elem1))`, but we have shortened the API for ease of use. The parameter, `b`, is the instance of the new `Block` object being created.

The context is an important tool as it allows an `Element` to act upon the values of other `Elements`. The context encapsulates the state of the current test case so that no changes to `Element` or `Block` instances need to be made during the generation process.

Once a block has been defined, a generator can be created with specific values to fuzz or no values to fuzz

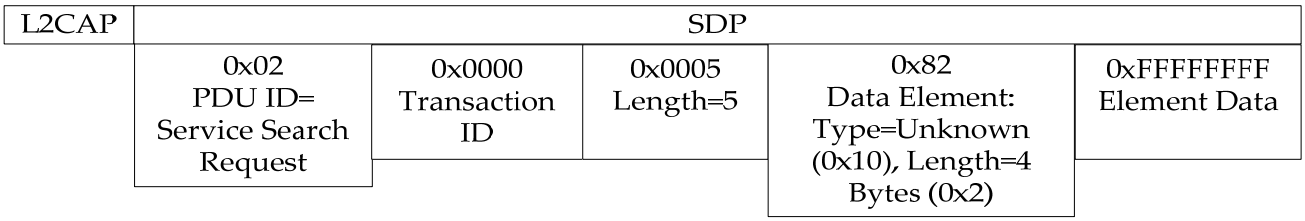
```
block = Fuzzit::Block.new do |b|
  b.IntElement(:elem1).has_length(3, :bytes)
  b.IntElement(:elem2).has_length(2, :bytes)
  b.LengthElement(:length).for(:string_elem).has_length(2, :bytes).has_endianness(:little)
  b.StringElement(:string_elem)
end
generator_with_default_values = block.fuzz_with()
generator_with_specific_value = block.fuzz_with(:string_elem => ["A"*100, "%s%s",
"\xff"*10000, "A"*100 + "\x00" + "A"*100])
```

Once we have a generator, it can be used to build all of the data structures we wish to fuzz with. Generators are able to index and describe the fuzzed data so that we can instantly rebuild the fuzzed data for a given test case and describe what value each `Element` took symbolically.

Using the precursor to Fuzzit, we discovered a vulnerability affecting the SDP protocol in the Bluetooth implementation of the Apple iPhone. The vulnerability has since been disclosed and fixed.

The SDP protocol specifies a primitive that includes a type descriptor and a size descriptor. Depending on the value of the size descriptor, the data for the primitive may have a fixed number of bytes or a variable number of bytes. If the size descriptor specifies a variable number of bytes, a fixed-width length field comes between the descriptor byte and the data. SDP also specifies that certain type descriptors are only allowed to have certain size descriptors.

Here is an example PDU that triggered the vulnerability:



Using an invalid type descriptor with a size descriptor specifying fixed length (4 bytes) data causes the Bluetooth server process to crash with:
Exception Type: EXC_BAD_ACCESS
Exception Codes: KERN_INVALID_ADDRESS at 0xFFFFFFFF

Here is a simplistic generator block that would generate this PDU:

```
sdp_block = Fuzzit::Block.new do |b|
  b.IntElement(:id).has_length(1, :byte).has_value(:service_search_request => 2)
  b.IntElement(:transaction_id).has_length(2, :bytes)
  b.LengthElement(:payload_length).for(:payload).has_length(2,:bytes)
  b.Block(:payload) do |p|
    p.BitBlock(:descriptor, 1) do |bb|
      bb.IntElement(:type).has_length(5,:bits).add_fuzz_values((0..0x1f).entries)
      bb.IntElement(:length_type).has_length(3,:bits).add_fuzz_values((0..7).entries)
    end
    p.LengthElement(:data_length).for(:data).fuzzes_length(0,1,2,4,8,16)
    p.IntElement(:data).fuzzes_length(1,2,3,4,5,6,7,8,16,100,1000,10000)
  end
end
```

Notice how in the offending PDU there is no data length field. By declaring the `:data_length Element` as having a fuzzable length with 0 being one of the lengths, that element will be omitted in a set of test cases. Also note that a fuzzable length corresponds to the width of an `Element`, not the content of that `Element`. How the content of `Elements` is fuzzed is explicitly specified by the `add_fuzz_values` call or by the default of fuzz values associated with the specific `Element` subclass.

Bringing it all together

After building creating a generator for a specific protocol or file format, the next step is to integrate it with an injection system, result analysis system, and optionally, a state management system. The Fuzzit framework has a convention for how all of these systems work together, so that components can be mixed and matched without any code needing to change. In the SDP example above, the generator was only useful without a robust state management system and result analysis system.

The examples shown above are purposefully simple to introduce Fuzzit and its basic usage. For more in-depth information on integrating a generator with other systems to create a full fuzzing system, advanced techniques for working with complex protocols, and examples of fuzzers implemented with Fuzzit, check out http://www.fuzzit.com.