carlpulley / Challenge11 http://honeynet.org/node/829
Submission for Honeynet Challenge 11 - Dive into Exploit

create pull request

Clone this repository (size: 18.1 MB): HTTPS / **SSH** / SourceTree

```
$ git clone git@bitbucket.org:carlpulley/challenge11.git
```

Answers

Notes:

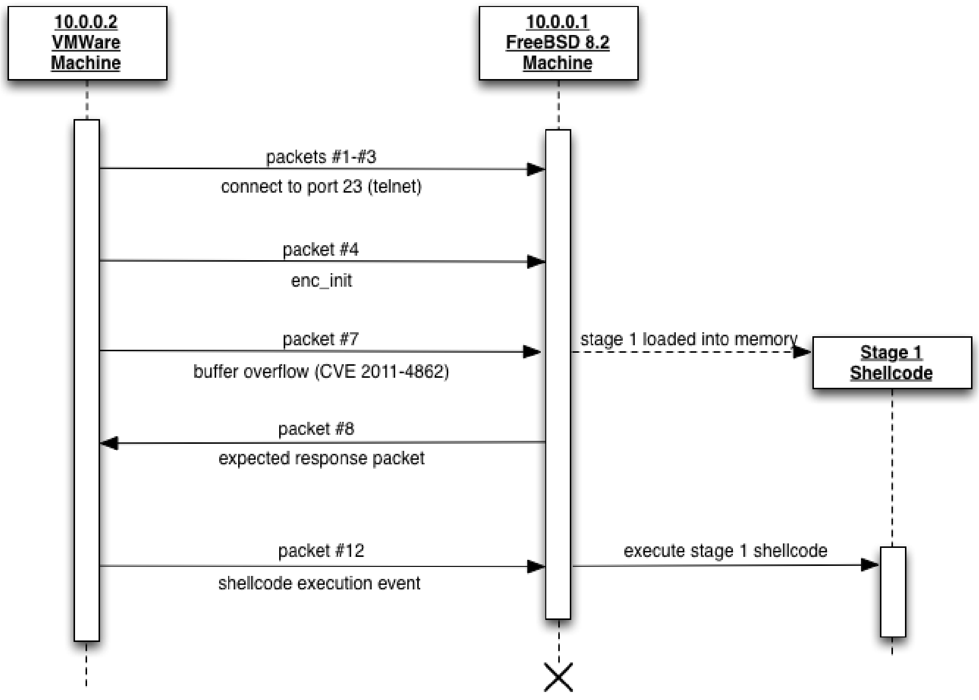
- a detailed analysis may be found at https://bitbucket.org/carlpulley/challenge11/wiki
- where possible, we have tried to solely use open-source tools - only when the specific need has arisen (e.g. in analysing the stage 2 payload) have we then switched to using the demo versions of commercial tools (i.e. the excellent IDA Pro)
- all computation for this work was performed on a 16 core, 2.4GHz Intel Xeon PowerMac, with additional support from the University of Huddersfield's HPC grid.

Question 1

10.0.0.1 is running a FreeBSD 8.2 operating system with a telnet service supporting encryption. Unfortunately, the telnet service is vulnerable to exploit CVE-2011-4862, a buffer overflow exploit that allows arbitrary code to be executed.

The attacker appears to use an exploit that is derived from the Metasploit FreeBSD Telnet Service Encryption Key ID Buffer Overflow?

The following sequence diagram provides an overview of the sequence of events that triggers this buffer overflow and then allows the (10.0.0.2) attacker's stage 1 code to execute:

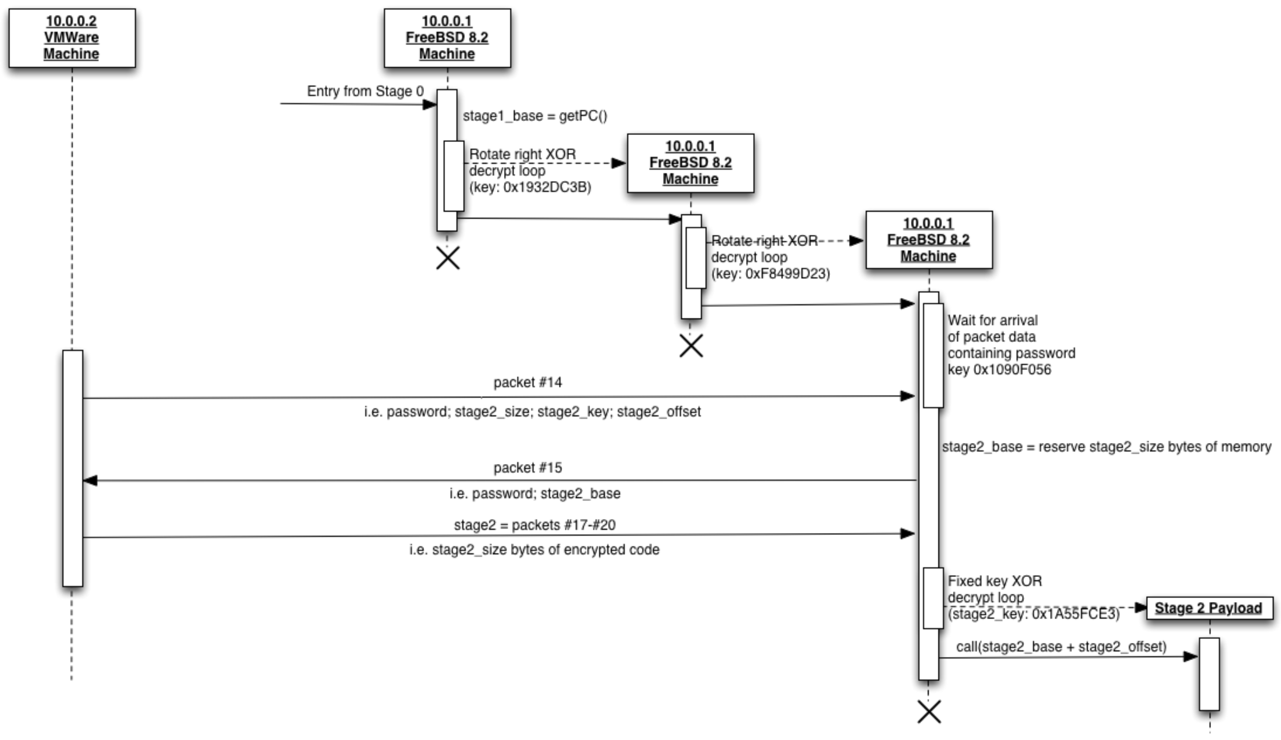

10.0.0.2 VMWare Machine
10.0.0.1 FreeBSD 8.2 Machine
packets #1-#3
connect to port 23 (telnet)
packet #4
enc_init
packet #7
buffer overflow (CVE 2011-4862)
stage 1 loaded into memory
Stage 1 Shellcode
packet #8
expected response packet
packet #12
shellcode execution event
execute stage 1 shellcode

Question 2

By saving an FPU environment record, the stage 1 payload executes a getPC sequence. This then leads into two separate XOR decryption loops that reveal the hard coded password #1090F056. This password is used to ensure that the stage 1 code is engaged with an expected server instance.

A series of exchanges between the server and the stage 1 payload now allow an XOR encrypted copy of the stage 2 payload, along with a starting offset and its decryption key, to be downloaded.

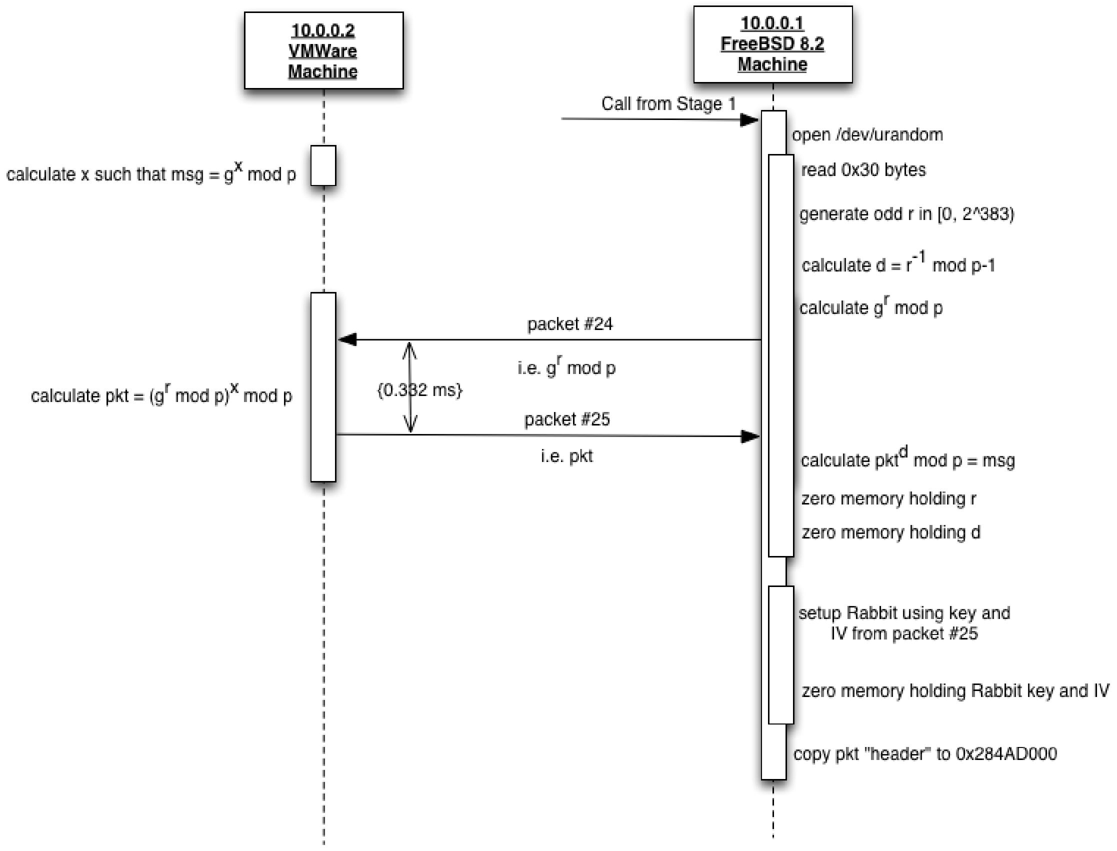
The following sequence diagram provides an overview of the sequence of events that download and execute the stage 2 payload:


10.0.0.2 VMWare Machine
10.0.0.1 FreeBSD 8.2 Machine
Entry from Stage 0
stage1_base = getPC()
Rotate right XOR decrypt loop (key: 0x1932DC3B)
10.0.0.1 FreeBSD 8.2 Machine
Rotate right XOR decrypt loop (key: 0xF8499D23)
10.0.0.1 FreeBSD 8.2 Machine
Wait for arrival of packet data containing password key 0x1090F056
packet #14
i.e. password; stage2_size; stage2_key; stage2_offset
stage2_base = reserve stage2_size bytes of memory
packet #15
i.e. password; stage2_base
stage2 = packets #17-#20
i.e. stage2_size bytes of encrypted code
Fixed key XOR decrypt loop (stage2_key: 0x1A55FCE3)
Stage 2 Payload
call(stage2_base + stage2_offset)

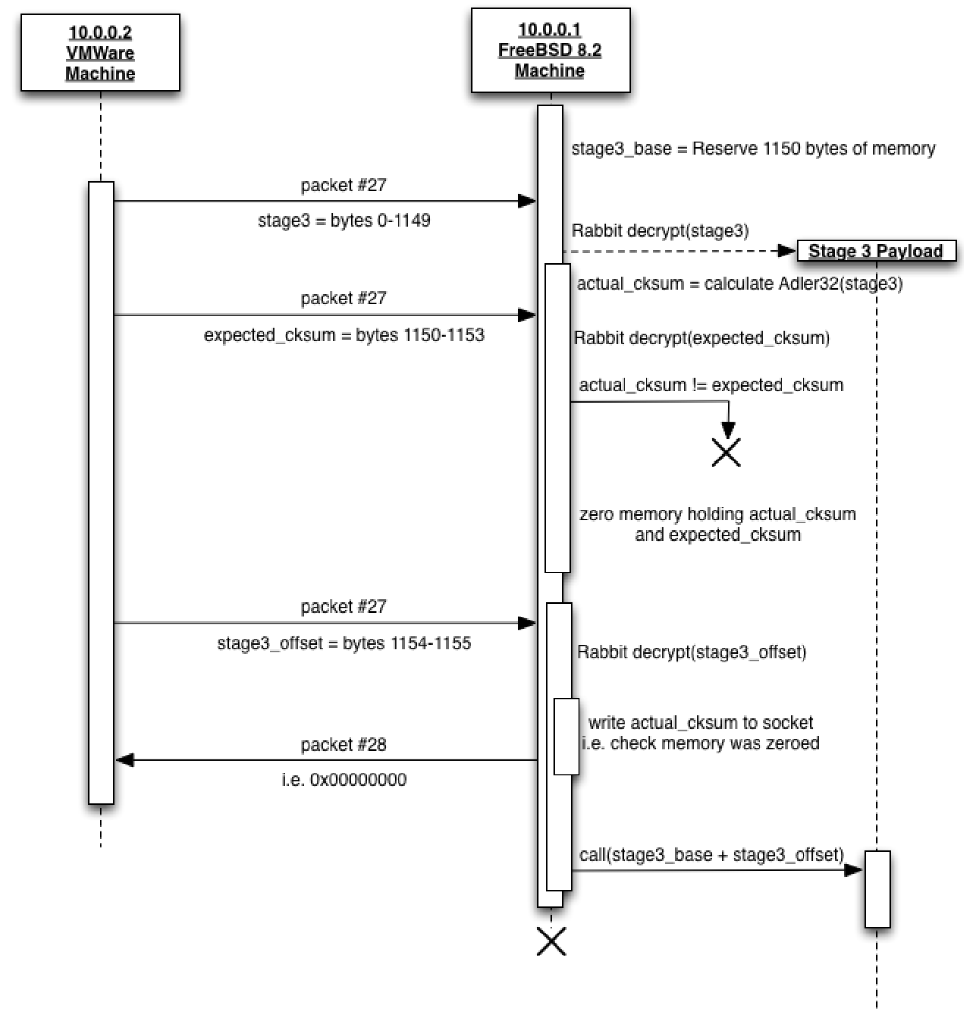
Question 3

The stage 2 payload may be separated into two phases:

- **phase 1:** generates a random session key that is communicated (via an exponentiation calculation) to the server and then, in the server's reply packet (another exponentiation calculation), used to communicate a shared initial state for the Rabbit stream cipher (this is extracted using the multiplicative inverse of the randomly generated session key):


10.0.0.2 VMWare Machine
10.0.0.1 FreeBSD 8.2 Machine
Call from Stage 1
open /dev/urandom
read 0x30 bytes
generate odd r in [0, 2^383)
calculate d = r^-1 mod p-1
calculate g^r mod p
calculate x such that msg = g^x mod p
packet #24
i.e. g^r mod p
{0.332 ms}
calculate pkt = (g^r mod p)^x mod p
packet #25
i.e. pkt
calculate pkt^d mod p = msg
zero memory holding r
zero memory holding d
setup Rabbit using key and IV from packet #25
zero memory holding Rabbit key and IV
copy pkt "header" to 0x284AD000

- **phase 2:** downloads (using the Rabbit stream cipher) and verifies (using an Adler32 checksum) the stage 3 payload, before executing it:

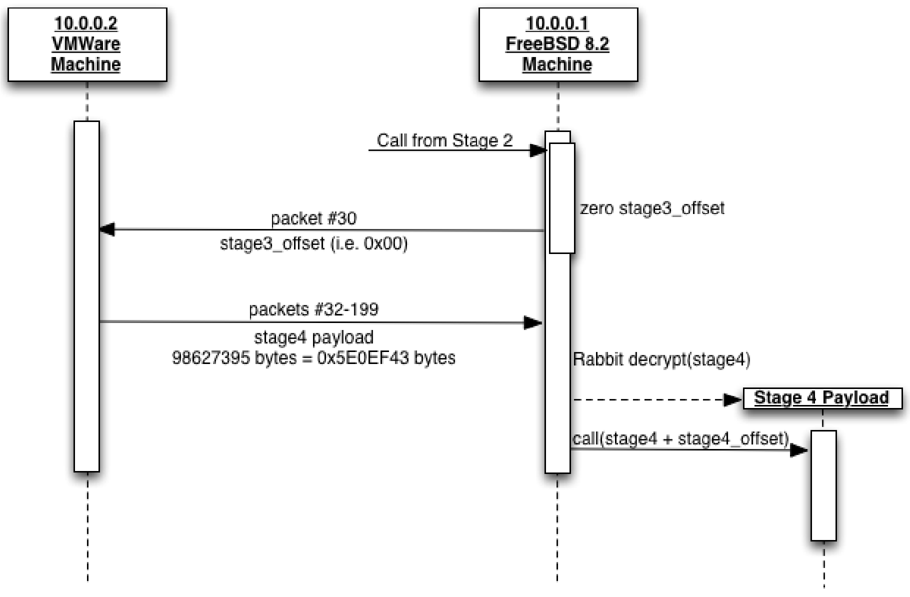

10.0.0.2 VMWare Machine
10.0.0.1 FreeBSD 8.2 Machine
stage3_base = Reserve 1150 bytes of memory
packet #27
stage3 = bytes 0-1149
Rabbit decrypt(stage3)
Stage 3 Payload
actual_cksum = calculate Adler32(stage3)
packet #27
expected_cksum = bytes 1150-1153
Rabbit decrypt(expected_cksum)
actual_cksum != expected_cksum
zero memory holding actual_cksum and expected_cksum
packet #27
stage3_offset = bytes 1154-1155
Rabbit decrypt(stage3_offset)
write actual_cksum to socket i.e. check memory was zeroed
packet #28
i.e. 0x00000000
call(stage3_base + stage3_offset)

Note: the Rabbit stream cipher is only used to encrypt communications from the server to 10.0.0.1. Other than packet #24, all communications from 10.0.0.1 to the server appear to be in plain text.

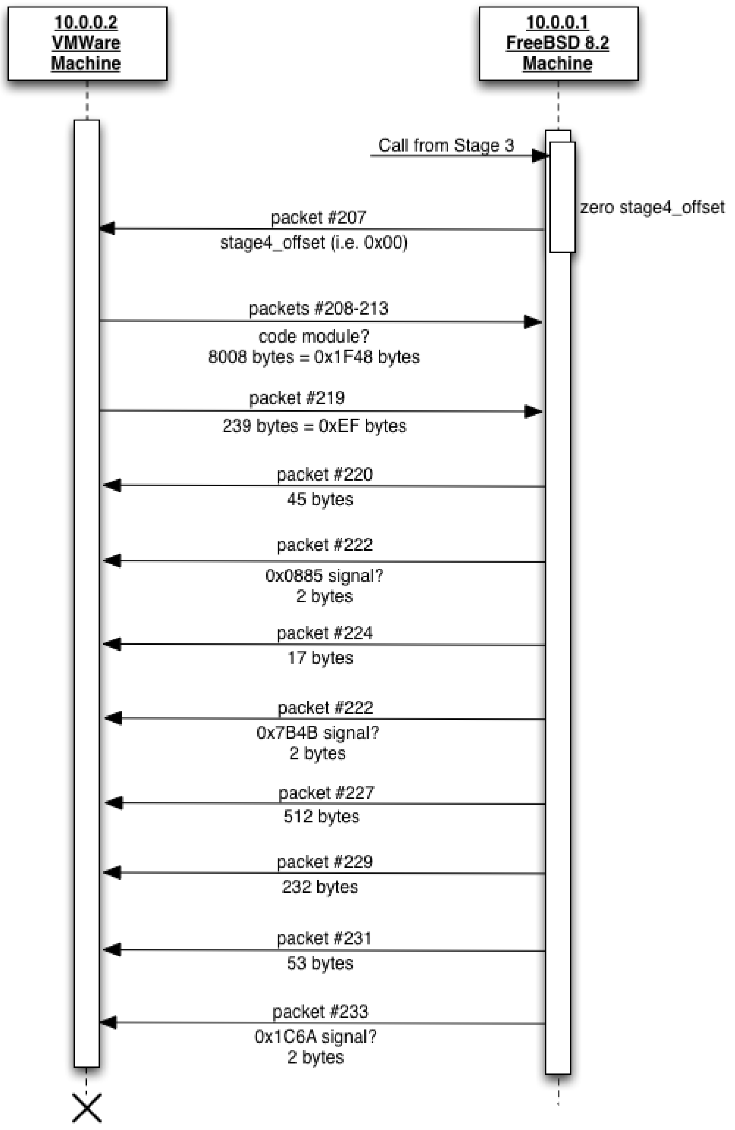
Question 4

Without cracking the stage 2 encryption, it has not been possible to answer this question accurately. However, based on known prior behaviour patterns, it is possible to make some reasonable estimations.

In examining the encrypted traffic between 10.0.0.1 and its server, we notice that, in packet #30, 10.0.0.1 communicates a null byte to the server. Based on known prior behaviour, we hypothesis that this packet marks entry into the stage 3 payload by writing back the stage 3 payload's entry offset (after it has been zeroed):



Examining the encrypted traffic between 10.0.0.1 and its server, we notice that, in packet #207, 10.0.0.1 communicates a null byte to the server. Again, based on known prior behaviour, we hypothesis that this packet marks entry into the stage 4 payload by writing back the stage 4 payload's entry offset (after it has been zeroed):


10.0.0.2 VMWare Machine
10.0.0.1 FreeBSD 8.2 Machine
Call from Stage 3
zero stage4_offset
packet #207 stage4_offset (i.e. 0x00)
packets #208-213 code module? 8008 bytes = 0x1F48 bytes
packet #219 239 bytes = 0xEF bytes
packet #220 45 bytes
packet #222 0x0885 signal? 2 bytes
packet #224 17 bytes
packet #222 0x7B4B signal? 2 bytes
packet #227 512 bytes
packet #229 232 bytes
packet #231 53 bytes
packet #233 0x1C6A signal? 2 bytes

Splitting our network packets using this observation, we infer that the stage 3 payload appears to just download 9862735 bytes (in packet's #32-#199) making up the stage 4 payload. As we hypothesis that an entry byte needs to be specified for the stage 4 payload, we note that this byte could be read from this downloaded data. Additionally, as we have previously witnessed Adler32 being used to perform checksumming of downloaded payloads, we presume that it is used again here. So, we speculate that:

- 4 bytes of the data is used for checksumming information
- and 1 byte is used to specify a starting offset.

Thus, we estimate that the stage 4 payload consists of the first 986270 downloaded bytes of packets #32-#199.

Without being able to reconstruct the random session key (by performing our discrete logarithm calculation), this is as far as we can reasonably go here with reconstructing the stage 4 payload.

Question 5

Unable to answer this question.

Question 6

- Avoid signature-based detection by obfuscating binaries
- Impede automated analysis with multi-stage loading and FPU based getPC instruction sequences
- Impede reversing efforts with:
  - use of randomly generated session keys
  - pre-computed discrete logarithms for an unknown factor base (providing coverage for all messages 10.0.0.2 wishes to send to 10.0.0.1)
- Modular structure/design to reduce information leaks by only loading code modules that are needed
- Secure coding practices (e.g. zeroing and repurposing of sensitive variables) to impede memory analysis
- Complex pointer arithmetic and memory manipulations to impede manual reversing efforts
- Use of standard algorithms and code libraries (e.g. use of a big num package and the Rabbit stream cipher) to simplify the codes development/production and to reduce probability of cryptographic programming/design errors.

This revision is from 2012-07-01 09:15