University of Amsterdam
System & Network Engineering

Offensive Technologies

Beating Metasploit with Snort

Authors:
Danny Groenewegen
danny.groenewegen@os3.nl

Marek Kuczyński
marek.kuczynski@os3.nl

Coordinator:
Jeroen van Beek
j.c.vanbeek@uva.nl

Abstract

Earlier research has shown that the effectiveness of Snort against the Metasploit Framework is very low. We made an attempt to improve the detection rate by automatically converting modules of the Metasploit Framework to Snort rules. Based on our analysis of the Metasploit modules we automated the process of generating Snort rules to detect payloads. Our tests have shown that the detection rates increased compared to the numbers based on earlier research. The results look promising, but need some further analysis. Especially encoders are still an unresolved issue.

July 2, 2011

CONTENTS

CHAPTER 1

INTRODUCTION

The Metasploit Framework[5] is an open source penetration testing solution. It is well known for its exploits to public security vulnerabilities. Penetration testers can use this as a powerful tool to detect potential vulnerabilities on a wide variety of different operating systems and software packages. As with many security tools it's not only used for legitimate purposes, but also for unauthorized activities by "hackers". Since the basic Metasploit Framework can be downloaded and used free of charge, it would be useful if these exploits could be detected and/or blocked from entering a network. The creation of rules for this purpose has not been attempted before, certainly not in an automated fashion.

A common detection and prevention mechanism is the use of Intrusion Detection Systems(IDS) and Intrusion Prevention Systems(IPS). These systems are used to detect and prevent malicious traffic on a network. A well known example of this is Snort[7], an IDS/IPS that combines the benefits of signature, protocol, and anomaly-based inspection. The effectiveness of Snort is defined by the network traffic rules that it contains. Different sets are available for Snort: rules from repositories included with the OS; the official rules of `www.snort.org` provided by the Sourcefire Vulnerability Research Team (VRT)[10]; community rule sets such as Emerging Threats[9]. However, earlier research[3] has shown that even a combination of these rule sets is not very effective against Metasploit exploits.

It would be beneficial if these rules could be generated automatically on every release of Metasploit, so that external attacks can be blocked before they reach a vulnerable system.

CHAPTER 2
RESEARCH QUESTION

2.1 Research question

Can Snort be enabled to automatically keep up with the latest exploits provided by Metasploit?

This question can be divided in sub-questions that cover several fields:

- What is the most effective way of detecting and blocking a malicious packet, is it by looking at exploits or payloads?
- Under which circumstances can we create Snort rules to detect Metasploit?
- What is the most effective way of creating Snort rules to detect malicious Metasploit traffic?
- What is the effect of false positive detections on Snort?

We will be focusing on the automatic generation of Snort rules, the possible performance impact caused by many rules won't be our main concern.

2.2 Previous research

Research has been done on the effectiveness of multiple Snort rule sets regarding the detection of Metasploit attacks [3]. The paper describes how the benchmark was done to find out the detection rate of Snort, but the authors do not improve the detection rates of Snort.

2.3 Approach

We will start with analyzing the exploits and payloads in Metasploit. The next step is looking into the possibilities of Snort rules, so we can define our approach for detecting malicious Metasploit traffic. To automatically extract relevant data from the Metasploit Framework we will look into two different approaches:

- Using the exploit and payload modules in the source tree of the framework

- Execute the exploits and capture the related traffic using tcpdump

After defining our detection approach and extracting the input data we will use a scripted approach to automatically create Snort rules.

CHAPTER 3

SNORT INTRUSION DETECTION

Snort[7] is an open source intrusion detection ("IDS") and prevention system ("IPS") . It can detect malicious or undesired network traffic and act upon it if needed. In order to protect a network, you can place a server running Snort either directly inline (for example in between a modem and a corporate network) or you can use it to only detect malicious traffic by placing it on a span or mirror port of a switch. In the latter configuration, Snort will analyze a copy of the packets transmitted, but it can't act upon an alert.

Snort analyzes traffic by the use of rules. These rules can analyze the content in various ways, either by looking at the type of traffic or the content of a packet. A basic example rule is listed below;

```
alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg:"mountd access";)
```

This rule will raise an alert whenever TCP traffic is sent to the internal 192.168.1.0 network. If a packet destined to this network contains the hex characters in the content tag, then the message shown in the msg tag is written to the Snort log.

There are many different parameters that can be used to pin-point or classify network traffic within Snort, but the example listed above shows the most common basic rule format. More details about the different rule options can be found in the extensive Snort manual [8].

During this project we made an attempt to generate Snort rules based on Metasploit. There are existing rulesets that focus on various forms of detection. The official Snort rules focus on many different detections, these get supplemented by rules created by multiple communities. Emerging Threats[9] focuses mainly on blocking known malicious IP addresses, for example known spammers or criminal organizations. These rules get updated frequently and can be freely downloaded from the Emerging Threats website.

CHAPTER 4

METASPLOIT FRAMEWORK

Metasploit is an open source framework used by IT professionals to test the security of their end-systems or network components. The framework is maintained by Rapid7[5], who maintains and expands the amounts of attacks available in the framework.

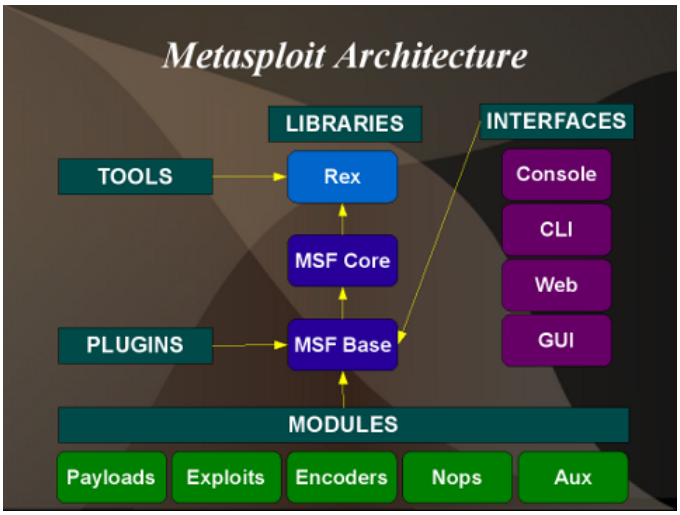

Figure 4.1: Metasploit Framework Architecture[6]

4.1 Metasploit Modules

The Metasploit Framework contains multiple types of modules which can be combined to attack a remote system. In this section a breakdown is given of these modules and a brief explanation is given on how they can be combined.

4.1.1 Exploits

Exploits are bits of code of commands that grant some form of access to a remote system. An example of this can be a malformed FTP request that eventually would allow you to execute

malicious code on the server. A lot of these exploits are the result of poor input sanitation, others make use of a permission elevation that the software developer or administrator did not take into account. In other words: exploits make use of known or unknown vulnerabilities in programs in order to gain access to a system. In the case of Metasploit, the exploits need to be combined with a predefined set of compatible payloads to get something done on the server.

4.1.2 Payloads

Once access to a system is granted through an exploit, a payload can be executed on the remote system. A payload is a piece of code that is used to execute commands on the targeted system, for example to add user credentials or additional permissions to a system. Staged payloads are also available: the first stage establishes a communication channel between the attacker and the target and retrieves the next stage(s) that will be executed on the remote host. Even more sophisticated payloads exist that make use of DLL injects on a host, known as Meterpreter in Metasploit terms.

4.1.3 Auxiliary

Auxiliary code consists of tools that assist in the finding of an exploit. Examples include port scanners and denial of service code, but also basic exploits that can reveal some information about the to-be-attacked system. Auxiliary modules can be considered exploits without payload, but their purpose is very diverse.

4.1.4 Encoders

One of the detection mechanisms used by IDSs/IPSs and anti-virus software is checking the signature of a packet or file. If the signature of a packet matches with the signature of a known exploit or payload, then a warning is given or an action is conducted. If the signatures is properly defined, then the data will still be detected when minor changes are made to it. Therefore, encoders have been created that can rearrange the order of some commands. This can be achieved by creating pointers throughout the package, and/or by filling it up with garbage. In some cases, these permutations can lead to the data not being detected as malicious any more, but it is still executable on the receiving node.

4.2 Module summary - relations between Metasploit modules

As you can see in Figure 4.2, Metasploit offers different modules that can be combined to hack a system. As already described above, the auxiliaries are used to gain information about a system or to crash something on the receiving side. They have no dependencies on other components in order to do this. The exploits are used to gain access to a system, but in general they won't do anything just by themselves. For this, they'll need a payload, which in essence is a bit of malicious code that is sent over the line after access to a system is gained with an exploit. Every exploit will specify the payloads that it is compatible with in the Metasploit Framework, so there is a limited set of matches that can be used. Last, in order to hide the presence of a payload to a virus scanner or IDS, they can be modified using an encoder, which will severely modify the file signature of the payload.

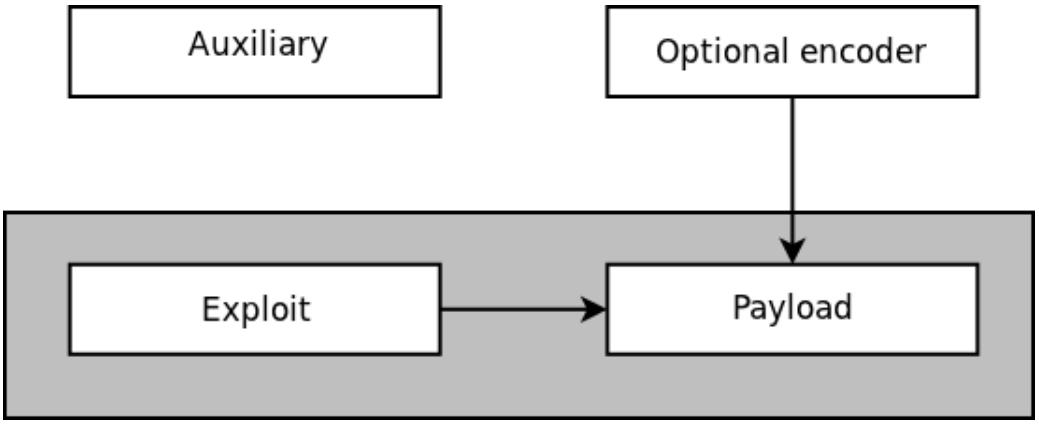

Figure 4.2: Relation of Metasploit modules

4.3 Specifying detection

We analyzed the components in order to find the best and most reliable detection method(s) for detecting payloads and exploits. Since both of these components must be present in a packet when hacking a system, it is enough to detect just one of the two halves with Snort. We analyzed the packet traces of several exploits and payloads combined and made the following observations:

- Exploits are responsible for the TCP session set-up, so it is possible that several packets are sent over the line before the actual malicious content is sent. It is very well possible that parts of the traffic generated by an exploit would classify as legitimate traffic, because in some cases it can be enough to gain access to a system just by changing a correct byte in an otherwise "legal" packet stream.
- It requires some effort to trick Metasploit into sending exploits over the line, because exploits facilitate some form of interaction between an attacker and victim. If the sending side does not receive a satisfying reply from the victim, then an error is raised. Even if this can be spoofed, its still hard to get reliable results from the traces between the two machines.
- Exploits have to be known to the world before any form of detection can take place, but payloads remain more static over time.
- Payloads can be generated very easily by Metasploit, and they consist of one packet. In some cases, this package becomes bigger than the MTU of the transport medium, but splitting up the signatures into smaller bits is enough to detect malicious traffic over the whole payload.

Therefore, the conclusion is that detecting exploits would be complicated and unreliable, while detecting the payloads is in theory possible in a reliable fashion.

CHAPTER 5

GENERATING RULES

This chapter gives an overview of the automated rule building process. We made several assumptions in this process, which are outlined in the next section.

5.1 Assumptions

- Signature lengths smaller than 32 bits get ignored and are not added to the definitive Snort ruleset. This value seems to be a good threshold to keep the amount of false positives down, although further research is required to determine an optimal minimum threshold.
- We generate each payload multiple times. By comparing the different versions of each payload we eliminate the following dynamic parts from the generated signatures:
  - A subset of all payloads include a random value. By generating each payload four times, we assume to have enough variation to eliminate the random values;
  - Payloads contain configurable options. For example the remote host, port or a command to execute. These values will differ, hence they should not be included in the Snort signatures. As an example, the values 6.6.6.6 and 9.9.9.9 are used for the remote host options. Their ascii representation is different so they can be extracted from the created signatures. These values are specifically chosen, because each bit is also different in binary representation: '0110 0110 0110 0110' and '1001 1001 1001 1001'.

Figure 5.1 illustrates the different parts of a payload. The signature used by Snort will only be based on the static code in the payload.

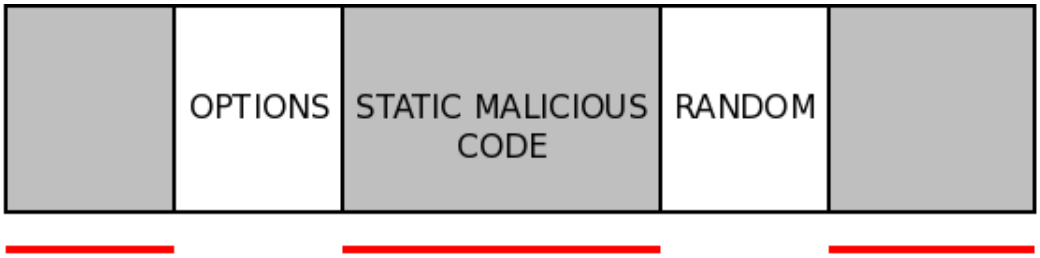

Figure 5.1: A payload consist of multiple parts.

5.2 Generating Rules

Taking into account the assumptions described above, we created a Ruby script to automatically convert all payloads to Snort rules. The full source code is can be found in appendix A. The following list gives a brief overview of this process:

1. Use the Metasploit Framework API to create an instance of the Framework
2. Generate each payload using two different option sets
3. Run a longest common substring algorithm[11] recursively on the two versions of each payload
   (a) Locate the longest static part using the LCS algorithm
   (b) Repeat for the remaining string left of the static part
   (c) Repeat for the remaining string right of the static part
4. Assemble the detected static part(s) into Snort content tags
5. Append the generated Snort rule to the output file

CHAPTER 6

ENCODERS

In the previous chapter a description is given of how signatures can be made from generated payloads. As shown in figure 4.2, these payloads can be obfuscated by the use of an encoder. This is done by reading the payload, inserting garbage instructions or jump references, resulting in a different signature than the original payload. The code execution on the host that is being attacked will have the same effect as the original non-encoded payload. Earlier research [1][4] has already shown that the detection of encoders is a difficult subject.

Metasploit comes with several built-in encoders, they can be used with multiple iterations and have an output in various different formats. However, the encoders are of varying quality: some of them will only change the capitalization while others will camouflage the code with NOP-sleds and jump references. The encoders can be mixed together, which means that you could do a couple of iterations with different encoders over a single payload.

We decided to look into detecting encoded payloads as well, in order to increase the Metasploit detection rate. We tried several different approached to detect encoded payloads, all of them are described below:

- We used a modified version of our rule generation script to detect static code parts among multiple iterations of each encoder using multiple payload. The output shows how often static code parts are present in the encoded payload. and how long the static parts are. Unfortunately, even one encoded payload may already deliver very variable results, since some payloads take random initialization vectors like the current time. Other encoders use polymorphic engines to randomly shift around blocks of code while keeping the algorithm intact.

- The source code of some Metasploit encoders was analyzed to check for leads. It appears that some encoders (i.e. shikata_ga_nai) do include static bits of code (which in example represent a jump instruction) but these signatures are too short to detect upon. Using them would most likely result in many false positives and this countermeasure would not apply for all encoders either.

- Another attempt was made with optimized recompiling of payloads. The output from a Metasploit encoder would be interpreted by an IDS, through which it would attempt to reconstruct the source code (or in this case, the plain payload instructions). If this is done in an optimized fashion, then the signature of the machine code should match the signature of the plain payloads. In order to analyze the code, a disassembler can be used. A disassembler can convert binary machine code into assembly code. By doing this, it is possible to analyze

what kind of assembly instructions are represented in the binary code that goes over the line. However, the code is still not optimized, so it will still contain the obfuscations and NOPs introduced by the encoder. Therefore, an optimization is required.

- This optimization can be delivered by a decompiler. A decompiler tries to interpret the code (like a de-assembler), after which it can optimize the code into something similar to the original code. The word "similar" already illustrates part of the problem with decompilers already; there is always some information lost or misinterpreted when decompiling code. It might be possible to check for routines or keywords in the decompiled code, but doing this reliably is a research field on its own.

CHAPTER 7

RESULTS

In the following two section we will describe our results of detecting payloads and encoders.

7.1 Payloads

To test our generated rules, we created a script that creates pcap packets for all payloads and counts how many are detected by Snort. These scripts can be found in Appendix C and D. The result shows that 109 of the total 222 payloads were detected by Snort using the generated rules. This is a detection rate of 49%.

We would suspect the detection rate to be higher, but we haven't looked into any of the payloads that were not detected. This is still an open issue and might be easy to improve.

7.2 Encoders

Analysis of the encoder sources did not lead to any concrete pointers. The detection of static code parts among multiple iterations of each encoder resulted in some static parts for a subset of all the encoders. However the detected parts were too short to be used as a signature for reliable detection of encoded payloads.

De-compilation and normalization of encoded payloads did not produce any usable result. However this is mainly caused by our lack of experience in the area of compilers. We had some pointers in our approach, but were not able to properly analyze this.

CHAPTER 8

CONCLUSIONS

During the research, several results were observed. In this chapter some of the findings are described and answers to the research questions are given. We created a script that generates all payloads and creates Snort of it. As a result, 49% of the payloads were detected successfully. This is already a big improvement compared to the default and community rule sets.

Looking back at the research question defined at the start of the research, the following conclusions can be made: *Can Snort be enabled to automatically keep up with the latest exploits provided by Metasploit?* Yes, but as described in chapter 4 we focused on the detection of payloads instead of exploits. Most, if not all, exploits require a payload to actually do something on the remote host. The code used to create Snort rules out of all available payloads makes use of the Metasploit API, which means that it can be updated once an update for Metasploit is released. As already shown in this chapter, the detection rate achieved with the generated Snort signatures is 49%.

Our approaches to detecting encoded payloads were not so successful. However, there might be some opportunities in the area of normalization and optimized recompiling.

8.1 Future work

- Further research is needed in the optimal signature size for Snort rules. Right now, the script provided with this paper takes the longest common substring and uses this to detect malicious packets. However, slight modifications to the payloads can already result in a signature mismatch. Therefor, smaller signatures would be more effective, since parts of them will and parts of them won't match the modified payload. However, the chance of false positives increases when the signatures get shorter.
- It would be interesting to see how many payloads can be detected when using a framework different from Metasploit. The payloads in Metasploit are optimized for efficiency, but it is unknown what the overlap is with other penetration suites.
- The research team was not able to make signatures out of the encoded payloads, as described in 6. A breakthrough could be achieved if encoded payloads could be normalized and compared to known payload signatures.

APPENDIX A

PAYLOAD RULE GENERATION SCRIPT

```
#!/usr/bin/env ruby

$:.unshift(File.join('','opt','framework-3.6.0','msf3','lib'))

require 'rex'
require 'msf/base'

# Initialize the simplified framework instance.
$framework = Msf::Simple::Framework.create(
    :module_types => [ Msf::MODULE_PAYLOAD, Msf::MODULE_ENCODER, Msf::MODULE_NOP ],
    'DisableDatabase' => true
)

$sid = 1000000

# read payload options file
payload_opts = Hash[File.read('payload_options.cfg').scan(/(.+?):(.+)/)]
payload_opts_diff = {}
# split up the diff options
payload_opts.each{ |x|
    a,b = x[1].split(",")
    payload_opts[x[0]] = a
    payload_opts_diff[x[0]] = b
}

def lcs_size(s1, s2)
   num=Array.new(s1.size){Array.new(s2.size)}
   len,posi,posj=0

   s1.scan(/./).each_with_index do |l1,i |
     s2.scan(/./).each_with_index do |l2,j |

        unless l1==l2
           num[i][j]=0
        else
          (i==0 || j==0)? num[i][j]=1 : num[i][j]=1 + num[i-1][j-1]
          if num[i][j] > len
            len = ans = num[i][j]
            posi = i
            posj = j
          end
        end
     end
   end
   [len,posi,posj]
end

#regexp filter for hex output
#require a minimum of two two character hex blocks seperated by whitespace
$regexpfilter = Regexp.new(/(\S{2}\s){3,}\S{2}/)

def common_payload(s1,s2)

```

```
54      lcs = lcs_size(s1,s2)
55
56      #string from size and position
57      #filter using regexp
58      if lcs[0] == 0
59          return []
60      else
61          lcs_s = s1[lcs[1]-lcs[0]+1..lcs[1]]
62          lcs_s = lcs_s[$regexpfilter]
63          if lcs_s.nil? || lcs_s.empty?
64              return []
65          end
66      end
67
68      # recursive left
69      if (lcs[1]-lcs[0] < 0) || (lcs[2]-lcs[0] < 0)
70          lcsleft = []
71      else
72          lcsleft = common_payload(
73              s1[0..(lcs[1]-lcs[0])],
74              s2[0..(lcs[2]-lcs[0])])
75      end
76      #recursive right
77      if (lcs[1] >= s1.length ) || (lcs[2] >= s2.length)
78          lcsright = []
79      else
80          lcsright = common_payload(
81              s1[(lcs[1]+1)..s1.length],
82              s2[(lcs[2]+1)..s2.length])
83      end
84
85      return lcsleft.push(lcs_s).push(lcsright)
86  end
87
88  # writes out the signatures to the snort rules file
89  def write_rule(name,content)
90      puts "Writing rule..."
91
92      $out.write('alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"'+name+'"; flow:
            established; ')
93
94  #    ERROR: ./snort/./rules/metasploit_payloads.rules(107) Line greater than or equal to
         32768 characters which is more than the parser is willing to handle.  Try splitting
         it up on multiple lines if possible.
95
96
97      content.each { |x|
98          if x.to_s.length >= 5
99              $out.write('content:"| '+x.strip+' |"; ')
100         end
101     }
102     $out.write("sid:#{$sid}; rev:1;)\n")
103     $sid = $sid + 1
104 end
105
106 #compare two strings for similar parts
107 # taking into account a maximum blocksize to speed things up
108 def compare_two(s1,s2)
109     contents = []
110     block_size = 1000
111     pos = 0
112
113     #find static parts using block_size splits
114     while pos < s1.length
115         #check if a full block is possible
116         if pos+block_size <= s1.length
117             contents.push(common_payload(s1[pos..(pos+block_size-1)], s2[pos..(pos+
                    block_size-1)]))
118             pos = pos+block_size
119         else
120             contents.push(common_payload(s1[pos..(s1.length-1)],s2[pos..(s2.length-1)]))
121             pos = s1.length
122         end
123     end
124
125     return contents.flatten
126 end
127
128 # open output file
129 $out = File.open('metasploit_payloads.rules', 'w')
```

```
130
131 # iterate all payloads
132 $framework.payloads.each_module { |name, mod|
133     payload_name = "#{name}"
134     puts "Generating "+ payload_name
135
136     begin
137         # generate payload four times with different option sets
138         payload = $framework.payloads.create(payload_name)
139         s1 = payload.generate_simple(
140             'Format'     => 'raw',
141             'Options'    => payload_opts,
142             'Encoder'    => nil)
143         payload = $framework.payloads.create(payload_name)
144         s2 = payload.generate_simple(
145             'Format'     => 'raw',
146             'Options'    => payload_opts_diff,
147             'Encoder'    => nil)
148         payload = $framework.payloads.create(payload_name)
149         s3 = payload.generate_simple(
150             'Format'     => 'raw',
151             'Options'    => payload_opts_diff,
152             'Encoder'    => nil)
153         payload = $framework.payloads.create(payload_name)
154         s4 = payload.generate_simple(
155             'Format'     => 'raw',
156             'Options'    => payload_opts,
157             'Encoder'    => nil)
158
159         puts "Converting payloads to hex format..."
160         s1 = Rex::Text.to_hex(s1,' ').to_s
161         s2 = Rex::Text.to_hex(s2,' ').to_s
162         s3 = Rex::Text.to_hex(s3,' ').to_s
163         s4 = Rex::Text.to_hex(s4,' ').to_s
164         puts "Extracting static payload parts..."
165
166         #extract the static parts from 4 generated payloads
167         contents1 = compare_two(s1,s2)
168         contents2 = compare_two(s3,s4)
169         contents = compare_two(contents1.join(' xx '),contents2.join(' yy '))
170
171         # check if static parts are found
172         if contents.length > 0
173             #write out the rule from static content parts
174             write_rule(payload_name,contents)
175         else
176             puts "No static parts found, skipping"
177         end
178
179     rescue
180         $stderr.puts payload_name
181         $stderr.puts "Error generating payload: #{$!}"
182         puts "#{$!.backtrace}"
183     end
184 }
185
186 $out.close
```

APPENDIX B

CONFIGURATION SCRIPT FOR PAYLOAD GENERATION

```
1  LHOST:6.6.6.6,9.9.9.9
2  DLL:/tmp/empty,/tmp/empty
3  BUNDLE:/tmp/empty,/tmp/empty
4  URL:/file1,/file2
5  CMD:something,nothing00
6  PEXEC:/tmp/empty,/tmp/empty
7  PXHOST:6.6.6.6,9.9.9.9
8  LPORT:6,9
```

APPENDIX C

CREATING PAYLOAD PACKETS

```
#!/usr/bin/env ruby

$:.unshift(File.join('','opt','framework-3.6.0','msf3','lib'))

require 'rex'
require 'msf/base'
require 'socket'

# Initialize the simplified framework instance.
$framework = Msf::Simple::Framework.create(
    :module_types => [ Msf::MODULE_PAYLOAD, Msf::MODULE_ENCODER, Msf::MODULE_NOP ],
    'DisableDatabase' => true
)

# read payload options file
payload_opts = Hash[File.read('payload_options.cfg').scan(/(.+?):(.+)/)]

puts payload_opts

# Start a local tcpserver to send payloads to
puts "Starting tcp server"
tcp_server = TCPServer.new('127.0.0.2',4444)

# iterate all payloads
$framework.payloads.each_module { |name, mod|

    puts "Generating "+ name

    begin
        # generate payload four times with different option sets
        payload = $framework.payloads.create(name)
        s1 = payload.generate_simple(
            'Format'    => 'raw',
            'Options'   => payload_opts,
            'Encoder'   => nil)
    rescue
        $stderr.puts "Error generating #{name}: #{$!}"
        puts "#{$!.backtrace}"
    end

    puts "Start tcpdump capture..."
    system('tcpdump -i lo -n -w ./captures/'+name.gsub(/\//,'-')+'.pcap&')
    sleep 1
    puts "Sending the payload..."

    # send the payload to a tcp socket
    t = TCPsocket.new('127.0.0.2','4444','127.0.0.1')
    t.write(s1)
    t.close

    sleep 2
    puts 'Saving tcpdump capture...\n'
    system('killall tcpdump')
```

```
54      sleep 1
55  }
56
57  puts "Done"
```

APPENDIX D

SNORT DETECTION RATE

```
1  #!/usr/bin/env ruby
2
3  # iterate all payloads
4
5  total = 0
6  total_detected = 0
7
8  $alertsfile = './snort/logs/alert'
9
10 Dir.glob('./captures/*.pcap').each { |file|
11     puts "Parsing "+file
12
13     puts "Starting Snort..."
14     system('snort -A fast -c ./snort/snort.conf -K none -k none -q -l ./snort/logs/ -r
           '+file)
15     sleep 0.5
16
17     detected = false
18     # read snort log file
19     if File.exists?($alertsfile)
20         puts "Snort detected:"
21         alerts = File.new($alertsfile,'r')
22         while (line = alerts.gets)
23             detected = true
24             puts line
25         end
26         alerts.close
27         File.delete($alertsfile)
28     else
29         puts "Snort detected nothing"
30     end
31
32     total = total + 1
33     if detected
34         total_detected = total_detected + 1
35     end
36 }
37
38 puts "Total:"+total.to_s
39 puts "Detected:"+total_detected.to_s
```

LIST OF FIGURES

BIBLIOGRAPHY

[1] Technology Flow. Metasploit encoding and antivirus detection. 2011.

[2] Dimitrios A. Glynos. Context-keyed payload encoding: Fighting the next generation of ids. 2010. Presented at AthCon 2010.

[3] Kevin de Kok Niek Timmers. Snorting metasploit. Technical report, University of Amsterdam, 2010. https://www.os3.nl/_media/2009-2010/students/niek_timmers/snorting_metasploit.pdf.

[4] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Real-world polymorphic attack detection using network-level emulation. pages 21:1–21:3, 2008.

[5] Rapid7. Metasploit. http://www.metasploit.com/, May 2011. version 3.7.1.

[6] Rapid7. Metasploit developer's guide. http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide, May 2011.

[7] Snort. Snort. http://www.snort.org/, May 2011. version 2.9.0.5.

[8] Snort. Snort official documentation. http://www.snort.org/docs, May 2011.

[9] Emerging Threats. Emerging threats. http://www.emergingthreats.net/, May 2011.

[10] Sourcefire Vulnerability Research Team^{TM} (VRT). Official rules of www.snort.org. http://www.sourcefire.com/, May 2011.

[11] Wikipedia. Longest common substring problem. http://en.wikipedia.org/wiki/Longest_common_substring_problem, 2011. [Online; accessed 25-May-2011].