

IDS Evasion Techniques and Tactics

Kevin Timm 2002-05-06

IDS Evasion Techniques and Tactics

by Kevin Timm

last updated May 7, 2002

Blackhats, security researchers and network intrusion detection system (NIDS) developers have continually played a game of point-counterpoint when it comes to NIDS technology. The BlackHat community continually develops methods to evade or bypass NIDS sensors while NIDS vendors continually counter act these methods with patches and new releases. Due to the inherent complexities involved in capturing, analyzing and understanding network traffic there are several common techniques that can be used to exploit inherent weaknesses in NIDSs.

Throughout this article we will explain basic evasion techniques as well as suggest fixes or what to look for in many of these attacks. For the purpose of this article we will consider evasion not only the process of totally concealing an attack but also a technique to disguise an attack to appear less threatening than it really is. These techniques can be divided into three specific categories, and one general category that is closely related to false negatives and poor implementation issues. The methods by which we can attack an IDS are through string matching weaknesses, session assembly weaknesses, and denial of service techniques.

Basic String Matching Weaknesses

Attacks based on basic string matching weaknesses are among the easiest to implement and understand. Signature-based IDS devices rely almost entirely on string matching and breaking the string match of a poorly written signature is trivial. Not all IDS devices are signature-based; however, most have a strong dependency on string matching. Because it is an open source tool, we will use [Snort](#) signatures to demonstrate applicable concepts.

To demonstrate the basics we are going to try to access the `/etc/passwd` file, which is a Unix file that contains user names, group memberships and associated shells. The following is an applicable snort signature.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-MISC
/etc/passwd";flags: A+; content:"/etc/passwd"; nocase;
classtype:attempted-recon; sid:1122; rev:1;)
```

This signature tries to match the string `/etc/passwd`. Due to this signature's simplistic nature, it

allows for the ability to alter the string very easily (regardless of the fact that it is HTTP traffic, we will touch more on that later). Examples of changing this string could be similar to `GET /etc//\//passwd`. Variants of attacks of this style could be `/etc/rc.d/././.\passwd` and countless other examples. This type of basic evasion is also relatively easy to defend against by carefully creating signatures that are general enough to catch most variants. To be fair to IDS developers, most of the more popular IDSs have powerful enough string matching capabilities that it is trivial to detect most variants of this. However, there still are times when signatures are poorly written, allowing basic string manipulation to evade the IDS.

The basic technique of breaking IDS string matching techniques leads to slightly more advanced techniques that are much more difficult to defend against, and just slightly more difficult to implement. An example here could be trying to read the `/etc/passwd` file through an interactive session such as telnet. It is not that uncommon for similar IDS signatures to exist in an attempt to enumerate patterns of misuse and backdoors. These signatures may look for common hacker tools' names, files, and programs. In our attempt to grab information from the `passwd` file in an interactive session, we can make use of command line interpreters. Through the use of command line interpreters we can obfuscate a simple `cat /etc/passwd` command into:

```
badguy@host$ perl -e
`$foo=pack("C11",47,101,116,99,47,112,97,115,115,119,100);
@bam=`/bin/cat/ $foo`; print"@bam\n";`
```

This in no way resembles the string of `/etc/passwd` that the IDS is trying to match. Defending against this is much more difficult because the IDS has to understand the interpreter and what is being passed to it. The IDS could simply alarm on command line use of an interpreter as being suspicious, but it could not easily match what is exactly being done with the interpreter.

By combining string manipulation techniques with character substitution techniques we can move into more complicated string obfuscation/manipulation. We can use the techniques similar to the above example in ordinary Web requests so there is no need for an interpreter. Hex encoding a url our request for `/etc/passwd` could resemble

```
GET %65%74%63/%70%61%73%73%77%64
or
GET %65%74%63/%70a%73%73%77d
```

To use ordinary string matching to try to catch every variant you would need over 1000 signatures for this string alone. This doesn't take Unicode into account. Unicode provides another means of character representation. (The problem Unicode presents for IDS is beyond the scope of this article;

however a detailed discussion is available in the SecurityFocus article [IDS Evasion with Unicode](#).) The HTTP scanning tool [Whisker](#) by RainForestPuppy has incorporated several of these methods. Here is a brief description of some the methods available in Whisker:

```
-I 1 IDS-evasive mode 1 (URL encoding)
-I 2 IDS-evasive mode 2 (./ directory insertion)
-I 3 IDS-evasive mode 3 (premature URL ending)
-I 4 IDS-evasive mode 4 (long URL)
-I 5 IDS-evasive mode 5 (fake parameter)
-I 6 IDS-evasive mode 6 (TAB separation) (not NT/IIS)
-I 7 IDS-evasive mode 7 (case sensitivity)
-I 8 IDS-evasive mode 8 (Windows delimiter)
-I 9 IDS-evasive mode 9 (session splicing) (slow)
-I 0 IDS-evasive mode 0 (NULL method)
```

A full description of these methods are available at [A Look At Whisker's Anti-IDS Tactics](#). All of the methods incorporated by Whisker involve the way the URI request is formatted with the exception of session splicing which will be discussed later.

IDS devices have handled most of these evasion techniques by better understanding the protocols and by doing all necessary protocol conversions before comparing the packet payload to the string. It is worth noting that all these character conversions can get expensive from a processing standpoint, so an IDS doesn't necessarily like to do extensive character conversions. To reduce the processing load developers may only run decoding on certain ports that are well known for specific traffic. This does provide the ability to evade the IDS when certain protocols are configured to listen on ports other than those associated with the IDS's well-known ports. Most recent IDS software releases handle the aforementioned techniques pretty well for traffic on well-known ports; however, systems that are a year old or older may not.

Polymorphic Shell Code

Polymorphic shell code is a more dangerous, recently developed evasion tactic that IDS devices cannot so easily defend against. Polymorphic shell code was developed by [K2](#) and is based on virus evasion techniques. This technique is limited to buffer overflows, and is much more effective against signature-based systems than anomaly or protocol analysis-based systems. As an example, let's look at what signatures Snort provides for an SSH CRC32 buffer overflows (this weakness is common to all signature-based systems not just Snort)

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32
overflow /bin/sh"; flags:A+; content:"/bin/sh"; reference:bugtraq,2347;
reference:cve,CVE-2001-0144; classtype:shellcode-detect; sid:1324; rev:1;)
```

```

alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32
overflow NOOP"; flags:A+; content:"|90 90 90 90 90 90 90 90 90 90 90
90 90 90")

```

As we can see, the first rule simply looks for the string `/bin/sh` with a destination port of 22 and the defined `$HOME_NET` destination network. The second signature looks for several x86 no-op characters with a destination port of 22 and `$HOME_NET` as the destination network. Polymorphic shell code uses several methods to evade string-matching systems. First (assuming x86 architecture for the example), it uses several other characters than x90 to perform no-op operations. Currently there are 55 replacements used on x86 (less on other architectures). These are combined in a pseudo-random manner (which can be weighted) to create the no-op section. A full list of available no-op replacements is available at <http://cansecwest.com/noplist-v1-1.txt>. The shell code itself is encoded using an XOR mechanism. This creates buffer overflows that do not resemble the above signatures and are resistant to standard string matches.

Detecting polymorphic shell code with a signature-based design is very challenging. Some ideas for detecting polymorphic code can be found in Next Generation Security Technologies' whitepaper [Polymorphic Shellcodes vs. Application IDSs](#). The basic premise of this research is that the polymorphic shell code can be detected with a reasonably high degree of accuracy by searching for a certain length pattern of no-op like characters. Recently, Dragos Ruiu released a [polymorphic pre-processor for Snort \(spp_fnord\)](#) that uses principles similar to those described above for detection. This configuration of this pre-processor is port and length dependent. So, for example, someone could configure the pre-processor to examine traffic on ports 80,21,23,and 53 for polymorphic evasion techniques. This would still leave all other ports such as 22 for SSH susceptible to polymorphic evasion techniques. With an effective polymorphic detection engine, effective device management can substantially reduce the likelihood of successful polymorphic evasion.

Session Splicing

The above evasion methods attempt to match a string within a packet without concern for session or how an attack may be delivered partially through multiple packets. Whisker has another network-level evasion method called session splicing. Session splicing divides the string across several packets as follows:

Packet number	Content
1	G

2	E
3	T
4	20
5	/
6	H

By delivering the data a few bytes at a time the string match is evaded. To handle this, the IDS either needs to watch the session and understand the session (which can be evaded in other ways such as the use of low TTL values) or detect the evasion technique through other techniques. The Snort rule to detect this is as follows.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-MISC whisker
space splice attack"; content:"|20|"; flags:A+; dsize:1;
reference:arachnids,296; classtype:attempted-recon; reference
```

This rule detects traffic destined to port 80 with the ack flag set, a space (hex 20) in the payload and a dsize of 1 (which will actually catch the first two bytes). While this accurately detects this tool, the method can be modified to evade the IDS. To defend against this it is possible to take the above Snort rule and extend it to look for HTTP requests that carry an abnormally small payload. This will most likely fire some false alarms, and is still able to be evaded under certain circumstances. To truly defend against this the IDS should fully understand the session, which is difficult and processor intensive. It should be noted that most current systems do reassemble sessions, but they usually have a certain amount of time that they reassemble the session for. This can be taken advantage of depending on the application, for example Apache on RedHat will time out a session in six minutes, but IIS on Win2K will keep a session alive for a very long time. It is possible to send one byte every 15 minutes and IIS will still consider this a valid session. The latest Snort releases have the ability to watch session for a long period of time and have the ability to watch for network level tricks such as low TTL values.

Fragmentation Attacks

Fragmentation attacks are similar to session splicing. Until recently, many devices didn't properly reassemble fragments before comparing them to the string. Fortunately, all current devices should do some reassembly before comparison. However, there are still several ways to evade an IDS through fragmentation. The problem with fragmentation reassembly is the IDS needs to keep the packet in memory and fully reassemble the packet before comparison to the string. The IDS also needs to understand how the packet will be reassembled by the destination host. The 1998 paper [Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection](#) by Thomas Ptacek and

Timothy Newsham details several network based fragmentation evasion methods as well as other network-based evasion techniques. Some of the possible evasion attacks based on fragmentation are fragmentation overlap, fragmentation overwrite, fragmentation timeouts and fragmentation techniques using network topology. These will be discussed in greater detail below.

Fragmentation overlap involves sending packets so one fragment overwrites data from a previous fragment therefore evading network detection. An example could be as follows:

```
Packet #1      GET x.idd
Packet #2      a.?(buffer overflow here)
```

The packets are assembled so that packet #2 overwrites the last byte of packet #1 so that when they are re-assembled on the destination host the string is GET x.ida?(buffer overflow).

Fragmentation overwrite is similar to overlap except that a complete fragment overwrites a previous fragment as such:

```
Packet #1      GET x.id
Packet #2      somerandomcharacters
Packet #3      a?(buffer overflow)
```

The way in which the host reassembles the fragments (i.e., whether it favors old or new fragments) will determine whether this is seen as a buffer overflow attempt or some other URI.

Fragmentation time-outs are dependent on how long the IDS holds the fragments in memory before discarding the packet. Most systems will time-out an incomplete fragment stream (in which it has received the first fragment) in 60 seconds. If the IDS does not hold on to the fragment for the full 60 seconds, it is possible to send packets as follows:

```
Packet #1      GET foo.id      MF bit set
Packet #2(59 seconds later)  a?bufferoverflow-here
```

If the IDS doesn't hold on to the initial fragment for a full 60 seconds it is possible to evade the IDS. Fortunately, current NIDS devices should be able to detect this if the device is configured correctly.

Fragmentation can be combined with some other network techniques such as using expiring TTL values. To use this, the host being attacked needs to be enough hops behind the IDS that the IDS

can see a packet that expires before reaching the destination host. The following is an example.

Packet Number	Payload	TTL
1	GET foo.id	> 2
2	evasion.html	1
3	a?bufferoverflow	> 2

In this example, the IDS will consider the request to be GET foo.idevasion.html; however, because the second packet will time-out before arriving at the host, the host will see the real attack, which is GET foo.ida?(bufferoverflow).

Fragmentation and Snort Signatures

Now let's compare these to some Snort signatures. The default Snort signature for the .ida buffer overflow will not catch any of these properly formatted fragmentation attacks (the exception being the time-out attack if the device is configured to use the frag2 pre-processor).

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-IIS ISAPI
.ida attempt"; uricontent:".ida?"; nocase; dsize:>239; flags:A+;
reference:arachnids,552; classtype:web-application-attack;
reference:cve,CAN-2000-0071; sid:1243; rev:2;)
```

However, Snort does have a signature to detect tiny fragmented packets, which will fire in most of the above examples.

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC Tiny
Fragments"; fragbits:M; dsize: < 25; classtype:bad-unknown; sid:522)
```

In some attacks this can be defeated as well. In the case of the ida exploit it doesn't really matter what URI is requested, so an attacker could front load the attack with garbage to prevent the tiny fragment from triggering. This could be done in a similar manner to that shown below.

```
Packet #1 GET reallylongstringtoevadedetect.it
Packet #2 da?(bufferoverflow)
```

It is important to note that these techniques are not entirely specific to Snort. [Cisco Secure IDS](#) reassembles fragments well and has alarms for overlap, multiple fragments claiming to be the same and several other fragmentation-related signatures. The problem is that, by default, they are set to a level 2 on the Unix director. In a standard implementation, these alarms may not be deemed serious enough to be analyzed. This is not a problem with the device in as much as it is a problem with the

personnel realizing the possible threat associated these alarms. The fragmentation alarms on the Cisco device, however, do not sufficiently log to allow for proper analysis. This poor logging can be somewhat attributed to the difficulties of understanding fragmented traffic. To further illustrate this, remember that except for the first fragment, no fragments have a TCP or UDP header, which means they have no information beyond the basic IP header.

The real problem with fragmentation, though, is much more complex, especially when using fragmentation with TTL and overwrites. The IDS needs to either understand the network on which it is implemented (such as whether this packet will time-out before getting to the host) or else traffic needs to be normalized for the IDS and the hosts on the network. The white paper by Vern Paxson and Mark Handley entitled [Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics](#) offers a good in-depth discussion on normalizing network traffic prior to IDS analysis.

The easiest fix for current systems is to verify that fragmentation time-outs are at least 60 seconds, have fragmentation alarms for several types of fragmentation anomalies, and have security personnel realize the potential threat associated with fragmentation alarms. To be fair to vendors and personnel, different fragmentation alarms can fire often and naturally for several types of traffic, which makes fully investigating and understanding these alarms very difficult. Dug Song recently released [Fragroute](#), a tool to check for many of the fragmentation vulnerabilities described by Newsham and Ptacek. Snort has recently implemented checks and methods to catch much of this network level trickery. A new official Snort release should contain many of these checks.

Denial of Service

A less civilized method of evasion is through denial of service. The denial of service can be either against the device or the personnel managing the device. Tools such as [Stick](#), [Snot](#) and several testing tools can be used to create a vast amount of alarms that can:

- consume the devices processing power and allow attacks to sneak by;
- fill up disk space causing attacks to not be logged;
- cause more alarms than can be handled by management systems (such as databases, ticketing systems, correlation engines, or alerting facilities);
- cause personnel to not be able to investigate all the alarms; and,
- cause the device to lock up

Since these tools are not necessarily state aware (and if they are, it reduces the amount of alarms they can send), IDS devices can become less susceptible if they are state aware however, UDP and

ICMP really have no associated state, so these are still good candidates for flooding (plus they are easily spoofed). Most devices are moving toward being state aware of certain protocols or ports. Once again, this has a processing cost associated with it and vendors try to wow everyone with impressive bandwidth stats.

While some of the aforementioned tactics are easily implemented, some do require a more seasoned or skilled attacker. There are however, several obvious methods that can be easily implemented and do not require a skilled attacker. These methods are more closely associated with false negatives and include encryption, phone lines, and so on.

The above really could be considered false negative conditions, but the end result of evasion and false negatives are the same.

Conclusion

This article touched on some of the more commonplace IDS evasion techniques. Traditional string matching weaknesses are becoming more difficult to evade; however, due to their inherent problems, network level evasion tactics such as splicing and fragmentation can still be successful. Vendors have recently made great strides toward defeating many of these evasion methods. More methods could be defeated if they were not so processing intensive. I would like to see an IDS architecture in which the user can turn on or off certain processing intensive modules easily, depending on the environment. Fortunately, processing power is increasing quickly, and if vendors are willing to sacrifice bandwidth, more prudent processing of events can be realized.

Kevin Timm is a Security Engineer at NetSolve Inc in Austin Tx.

[Privacy Statement](#)

Copyright 2006, SecurityFocus