Network Intrusion Detection Signatures

This is article on understanding and developing signatures for network intrusion detection systems. In this article we will discuss the basics of network IDS signatures and then take a closer look at signatures that focus on IP, TCP, UDP and ICMP header values. Such signatures ignore packet payloads and instead look for certain header field values or combinations of values. By learning about network IDS signatures, you'll have more knowledge of how intrusion detection systems operate, and you'll have a better foundation to write your own IDS signatures.

Signature Basics

A network IDS signature is a pattern that we want to look for in traffic. In order to give you an idea of the variety of signatures, let's quickly review some examples and some of the methods that can be used to identify each one:

- Connection attempt from a reserved IP address. This is easily identified by checking the source address field in an IP header.
- Packet with an illegal TCP flag combination. This can be found by comparing the flags set in a TCP header against known good or bad flag combinations.
- Email containing a particular virus. The IDS can compare the subject of each email to the subject associated with the virus-laden email, or it can look for an attachment with a particular name.
- DNS buffer overflow attempt contained in the payload of a query. By parsing the DNS fields and checking the length of each of them, the IDS can identify an attempt to perform a buffer overflow using a DNS field. A different method would be to look for exploit shellcode sequences in the payload.
- Denial of service attack on a POP3 server caused by issuing the same command thousands of times. One signature for this attack would be to keep track of how many times the command is issued and to alert when that number exceeds a certain threshold.
- File access attack on an FTP server by issuing file and directory commands to it without first logging in. A state-tracking signature could be developed which would monitor FTP traffic for a successful login and would alert if certain commands were issued before the user had authenticated properly.

As you can see from this list, signatures range from very simple – checking the value of a header field – to highly complex signatures that may actually track the state of a connection or perform extensive protocol analysis. In this article, we'll be looking at some of the simplest signatures and discussing the complexities involved in developing even the most basic signature. Note that signature capabilities vary greatly among IDS products, so some of the techniques described here may not be possible to implement in the IDS that you use. For example, some network IDS products provide little ability to customize existing signatures or write your own, while other IDS products give you the ability to customize all their signatures and write almost any signature you can think of. Another important factor to consider is that some IDS products can only check certain header or payload values, while other products can give you the data from any portion of any packet.

What Functions Do Signatures Serve?

This may seem like an obvious question, but what is the purpose of an intrusion detection signature? The answer is that different signatures have different goals. The obvious answer is that we want to be alerted when an intrusion attempt occurs. But let's take a moment to think about other reasons why we might want to write or modify a signature. Perhaps you're seeing some odd traffic on your network and you want to be alerted the next time it occurs. You've noticed that it has unusual header characteristics, and you want to write a signature that will match this known pattern. Or perhaps you are interested in configuring your IDS to identify abnormal or suspicious traffic in general, not just attacks or probes. Some signatures may tell you which specific attack is occurring or what vulnerability the attacker is trying to exploit, while other signatures may just indicate that unusual behavior is occurring, without specifying a particular attack. It will often take significantly more time and resources to identify the tool that's causing malicious activity, but it will give you more information as to why you're being attacked and what the intent of the attack is.

Header Values

Now that we've taken a quick look at signature types, let's focus on a simple signature characteristic: header values. Some header values are clearly abnormal, so they make great candidates for signatures. A classic example of this is a TCP packet with the SYN and FIN flags set. This is a violation of RFC 793 (which defines the TCP standard), and has been used in many tools in an attempt to circumvent firewalls, routers and intrusion detection systems.

Many exploits include header values that purposely violate RFCs, because many operating systems and applications have been written on the assumption that the RFCs would not be violated and don't perform proper error handling of such traffic. Also, many tools either contain coding mistakes or are incomplete, so that crafted packets produced by them contain header values that violate RFCs. Both poorly written tools and various intrusion techniques provide distinguishing characteristics that can be used for signature purposes.

This all sounds great – but of course, there's a catch. Not all OSs and applications completely adhere to the RFCs. In fact, many have at least one facet of their behavior that violates an RFC. Also, over time, protocols may implement new features that are not included in an RFC. And new standards emerge over time, which may "legalize" values that were previously illegal; RFC 3168, for Explicit Congestion Notification (ECN), is a good example of this. So an IDS signature based strictly on an RFC may produce many false positives. Still, the RFCs make a great basis for signature development, because so much malicious activity violates RFCs. Because of RFC updates and other factors that we'll discuss later, it's important to review and update existing signatures periodically.

Although illegal header values are certainly a fundamental component of signatures, legal but suspicious header values are at least as important. For example, alerting on connections to suspicious port numbers such as 31337 or 27374 (often associated with Trojans) may provide a quick way of identifying Trojan activity. Unfortunately, some normal, benign traffic may happen to use the same port numbers. Without using a more detailed signature that includes other characteristics of the traffic, you won't be able to determine the true nature of this traffic. Suspicious but legal values such as a port number are best used in combination with other values.

Identifying Possible Signature Components

The best way to understand the issues in developing a signature based on header values is to walk through an example. synscan is a tool that is widely used in scanning and probing systems. synscan activity was seen in unusually large amounts on the Internet in early 2001 because its code was used to create the first phase of the Ramen worm. This activity makes a great example, because the packets have several distinguishing characteristics. Here are some of the IP and TCP header values that were present in Ramen worm packets during the first stage of the worm's spread. (Note that my box was configured to silently drop unsolicited traffic, so I only saw the first packet of each attempt.)

- Various source IP addresses
- TCP source port 21, destination port 21
- Type of service 0
- IP identification number 39426
- SYN and FIN flags set
- Various sequence numbers set
- Various acknowledgment numbers set
- TCP window size 1028

Now that we know what the characteristics of the synscan packet headers are, we can start to consider what a good signature would be. We're looking for values that are illegal, unusual or suspicious – in many cases, these characteristics correspond to the vulnerabilities that the attacker is trying to exploit, or a particular technique that the attacker uses. Packet values that are completely normal don't make good signature characteristics by themselves, although they are often included to limit the amount of traffic that we study. For example, we would include the normal IP protocol value of 6 for a protocol, so that we only check TCP packets. But other characteristics that are completely normal, such as the type of service set to 0, are much less likely to be helpful in signature development.

Certain characteristics of the synscan packets are anomalous and could be used in signatures:

- Having only the SYN and FIN flags set is a well-known sign of malicious activity.
- Another sign that these are crafted packets is that the acknowledgment number has various values but the ACK flag is not set. When that flag isn't set, the acknowledgment number should be set to 0.
- Another suspicious characteristic is that the source and destination ports are both set to 21, usually associated with FTP servers. When these ports equal each other, we say that they are reflexive. With a few exceptions (such as certain types of NetBIOS traffic), we should normally not see these two values equal to each other. Reflexive ports don't violate TCP standards, but in most cases they are unexpected and unusual. In normal FTP traffic, we would expect to see a high port number (greater than 1023) as the source and port 21 as the destination.

So far we've identified three likely signature elements: the SYN and FIN flags set, the acknowledgment number set to a non-zero value without the ACK flag set, and reflexive ports set to 21. There are two more items that are noteworthy: the TCP window size, which is always set to 1028, and the IP identification number, which is set in all the packets to 39426. Normally, we would expect the TCP window size to be larger than 1028; although this value is not terribly abnormal, it's more than a coincidence that it's the same in all of these. Likewise, the IP identification number is set in all packets to 39426. The IP RFC specifies that this value should vary among packets, so the constant value is very suspicious.

Choosing a Signature

Because we've identified five potential signature elements, we have many different options for developing a header-based signature, because a signature could include any one or more of these characteristics. A simple signature would be packets with only the SYN and FIN flags set. Although this would certainly be a good indicator of likely malicious activity, it doesn't give us any idea why this activity occurred. Remember that SYN and FIN have traditionally been used together to circumvent firewalls and other devices, so their presence could indicate that scans are being conducted, information gathered or attacks launched. So a signature based on SYN and FIN only may be too simple to meet our goals.

However, a signature based on all five suspicious characteristics may be too specific. Although it would provide much more precise information about the source of the activity, it would also be far less efficient than a signature that only checks one header value. Signature development is always a tradeoff between efficiency and accuracy. In many cases, simpler signatures are more prone to false positives than more complex signatures, because simpler signatures are much more general. But more complex signatures may be more prone to false negatives than simpler signatures, because one of the characteristics of a tool or methodology may change over time.

Let's assume that one of our signature's goals is to determine the tool being used. So besides the SYN and FIN flags, what other attributes would be best to examine? Well, the reflexive port numbers are suspicious, but they don't provide a specific enough signature, as many tools use them, as well as some legitimate traffic. The ACK value set without an ACK flag is clearly invalid and might make a good signature on its own, as well as paired with SYN and FIN. The window size of 1028, although a bit suspicious, could occur naturally. So could the IP identification number of 39426. We could develop several signatures that use various combinations of these characteristics. It's often unclear as to what the best signature is, especially because the optimal signature may vary among environments and is also likely to change over time.

Conclusion

In the next article in this series, we'll decide which attributes we should use for our synscan signature and then evaluate the effectiveness of that signature against more synscan traffic. We'll further investigate the merits of general signatures as opposed to specific signatures. We'll also continue to look at the role that IP protocol header values play in signature development.

Now we will continue our discussion of IP protocol header values in signatures by closely examining some signature examples. Although it may be relatively easy to develop a signature that matches a particular type of traffic, it will likely cause unexpected false positives and false negatives. Signatures must be carefully developed and tested in order to create a signature set that is highly accurate, yet is also as efficient as possible.

Evaluating the Effectiveness of a Signature

In the previous article, we looked at the characteristics of the packets sent by the synscan tool (as implemented in the Ramen worm) and identified traits that were unusual or suspicious, or that violated standards. We then tried to determine which of these traits might make a good signature. Based on those characteristics, let's create a signature that will look for all three of the following attributes in each TCP packet:

- Only the SYN and FIN flags set
- IP identification number 39426
- TCP window size of 1028

The first characteristic, having only the SYN and FIN flags set, is too general to be used alone to identify synscan activity. Even though the second and third items can and do occur in legitimate traffic, the odds of both occurring in the same packet are very low, so it's reasonable to use them along with the SYN and FIN flags to create a detailed signature. Adding the other synscan attributes would not significantly improve the accuracy of this signature, although they would increase the amount of resources needed to identify it. So do these three characteristics make a strong signature for the synscan tool? Well…yes and no. They constitute a great signature for that particular version of the synscan tool; however, there have been many versions of the synscan tool, and it's certainly likely that some of them have different packet header characteristics.

A much more interesting question is, will the signature that we've developed be able to detect variants on the synscan tool – that is, other tools that have been developed using the synscan tool? In the case of the Ramen worm, the answer is yes; but what about other such traffic? Will our signature be able to find synscan variants? It's often shortsighted to develop just a signature that is specific to a particular implementation of an attack; by using a combination of general and specific signatures, you can often create a much better overall solution. An intrusion detection signature set is much more valuable if it can detect not only known attacks, but also future and unknown attacks. Let's examine this concept in more detail.

Applying the Signature to More Anomalous Traffic

Within weeks of the Ramen worm/synscan activity that our signature is based on, I received some additional scanning attempts that had a similar signature, with a few important differences:

- Instead of only the SYN and FIN flags being set, only the SYN flag was set. This is a normal TCP packet characteristic.
- The TCP window size was always 40 instead of 1028. 40 is an unusually small window size for an initial SYN packet and is certainly much less likely to be seen in a normal packet than 1028.
- The reflexive port number was 53 instead of 21. Old versions of BIND used reflexive ports for certain operations, but newer versions of BIND don't, so we wouldn't expect to see reflexive port TCP 53 very often.

Because there were so many similarities between the first set of packets and the second, we can reasonably conclude that the second set of packets were either generated by a different version of synscan or by a different tool that was based on the synscan code. Obviously, the synscan signature we developed won't catch this variant, because two of the three characteristics we were checking for have changed. So now what do we do? Well, we could create an additional synscan signature that would match the variant. Or we could adjust our goals, and focus on alerting on generally abnormal behavior rather than on particular implementations of tools. Or we could do both, creating some specific signatures that alert on the exploit or scanner implementations and some general signatures that look for basic anomalies.

General signatures that might be useful for this set of traffic are:

- TCP packet with an acknowledgement value set to a non-zero number, but the acknowledgment flag is not set
- TCP packet with only the SYN and FIN flags set
- TCP packet with the initial TCP window size below a certain value (which would include 40)

Two of these three general signatures would match for both types of packets that we've looked at so far.

Identifying Traffic from Yet Another Variant

Weeks after seeing the apparent synscan variant that we just reviewed, I received some packets with almost the same characteristics. In fact, the only difference was that the IP identification number was no longer static. One likely explanation for this is that this third group of packets was made by a variant of the variant of synscan that made the second set of packets. It's possible that someone took parts of the synscan variant's code and created another tool from it. It's also possible that someone fixed the synscan code so it would no longer have a static IP identification number. This makes synscan traffic less distinctive and more stealthy from an intrusion detection system point of view by eliminating the IP identification number as a potential signature characteristic.

By this point, it should be obvious that the original synscan signature that we developed would not match this traffic. In fact, none of the three components of the original signature are present in this variant. But if you look at the three general signatures we developed, you'll see that two of the three would match this traffic too. Even though the characteristics of the traffic are changing, we can still identify it as anomalous through the use of more general signatures. This reinforces the importance of using general signatures that look for individual anomalies in traffic, rather than relying solely on highly detailed signatures that match particular attacks or exploits.

If you still wanted to specifically identify traffic from these variants, you'd want to create additional detailed signatures that would match them. Again, whether you should do this or not depends on what your goal is. Many people couldn't care less which tool is being used to attack them; they're only interested in knowing that something bad is happening. But in many environments, there's a strong need to know the likely intent of the attacker. If you simply alert on generally anomalous activity, without understanding the potential significance of that activity, you won't know what the attacker is trying to do. By basing signatures on in-depth research of potential attacks and vulnerabilities, signature sets can be developed that will include general and specific signatures in order to identify the likely source or purpose of a variety of malicious traffic.

Final Notes on Header Values

In the examples in this article, we've seen how several header values can be used to create network IDS signatures. In general, some of the most commonly used header-related signature elements are:

- IP addresses (particularly reserved, non-routable, and broadcast addresses)
- Port numbers that should not be in use (especially well-known ports for particular protocols and Trojans)
- Unusual packet fragmentation
- Particular TCP flag combinations
- ICMP types/codes that should not normally be seen

Of course, any header value can be used in signatures; these items are just more likely to be used than most others.

An issue that we haven't yet addressed is which packets we want to check. If we're using a signature that is based on header values, which packets should we be checking – all of them or some of them? Well, it depends. Because ICMP and UDP packets are connectionless, odds are that under most circumstances, you'll want to check each of them. Since TCP is connection-oriented, sometimes you can just check the first packet in a connection. For example, certain characteristics such as addresses and ports will remain constant in all packets in the connection, so they can just be checked once. Other characteristics such as TCP flags should be different among the packets in the session, so if you're looking for particular flag combinations, you'd want to check for those in every packet. Of course, the more packets you check, the more time and resources it will take.

You may be wondering why we have been focusing on values in IP, TCP, UDP and ICMP headers, and have not mentioned others such as DNS. The reason has to do with the way that packets are structured. Remember that TCP, UDP and ICMP are all IP protocols, so their headers and payloads are located inside the IP packets' payload portion. In order to get TCP header data, for example, we first need to parse the IP header so we can determine that the payload is TCP. Other protocols such as DNS are contained inside UDP and TCP packet payloads, so we have to go through two levels of parsing (IP and UDP or TCP) in order to get to them. Also, it takes far more work programmatically to decode many of these protocols, as compared to the relatively simple structure of TCP, UDP and ICMP headers. This decoding is what really differentiates the protocols, as many network intrusion detection systems lack the ability to do much protocol analysis. We'll look much more closely at protocol analysis in future articles in this series.

Conclusion

During this article and the previous one, we have learned some of the basic concepts of network intrusion detection signatures, particularly the difficulties of signature development. Signatures are moving targets of sorts, because attackers can easily modify tools, which causes the characteristics of their traffic to change. By utilizing two approaches - using specific signatures that look for traffic from specific tools or particular exploits, and general signatures that just look for anomalies in traffic – an intrusion detection system can identify known and unknown attacks. Various intrusion detection systems offer vastly different signature sets, as well as signature writing capabilities and features.

We've also looked at the importance of IP protocol header values, particularly TCP, UDP and ICMP, in network intrusion detection signatures.

We examined the use of IP protocol header values, particularly TCP, UDP and ICMP, in network intrusion detection signatures.

Now we will continue our discussion of signatures by studying the area of protocol analysis, focusing on the examination of values within TCP and UDP payloads. Network intrusion detection using protocol analysis-based signatures is very effective in detecting both known and unknown attacks involving protocols such as DNS, FTP, HTTP and SMTP.

The Basics of Protocol Analysis

The first two articles in this series focused on developing network intrusion detection signatures using values in IP, TCP, UDP and ICMP headers. Now we want to look at signatures that examine the payloads within TCP and UDP packets, which contain other protocols.

It’s important to understand that a protocol such as DNS is contained within TCP or UDP, which itself is contained within IP. So we first decode a packet’s IP header information, which will tell us whether its payload contains TCP, UDP or another protocol. If the payload is TCP, for example, we then need to process some of the TCP header information within the IP payload before we can access the TCP payload. DNS data is contained within UDP and TCP payloads.

Because intrusion detection systems are normally most interested in IP, TCP, UDP and ICMP characteristics, they normally are able to decode some or all of their headers. However, only more advanced intrusion detection systems are capable of performing protocol analysis. Such IDS sensors perform full protocol decoding for DNS, HTTP, SMTP and other widely used protocols. Due to the complexity of decoding each protocol, as well as the sheer number of protocols in wide use, protocol analysis requires much more advanced IDS capabilities than simpler signature techniques, such as “packet grepping”. IDS sensors that perform packet grepping simply look for a particular string or sequence of bytes within a packet; the sensor has no real knowledge of the protocol that it is examining, so it can only identify malicious activity that has an obvious, simple signature.

The term “protocol analysis” means that the IDS sensor understands how various protocols work and closely analyzes the traffic of those protocols to look for suspicious or abnormal activity. For each protocol, the analysis is based not only on protocol standards, particularly the RFC’s, but also on how things are implemented in the “real world”. Many implementations violate protocol standards, so it is very important that signatures reflect how things are really done, not how they are ideally done, or many false positives and negatives will occur. Protocol analysis techniques observe all traffic involving a particular protocol and validate it, alerting when the traffic does not meet expectations.

At this point, it is probably not yet clear to you why protocol analysis is a better technique than simple packet grepping. The problem with packet grepping signatures is that they are usually written to look for a particular known exploit. In most cases, you can’t write a packet grepping signature to identify an unknown attack or a variation on a known attack, or to identify general attacks that are attempting to exploit a particular vulnerability. Therefore, packet grepping signatures tend to have limited effectiveness in identifying attacks. Protocol analysis gives you the ability to look for any activity that violates standards or expected behavior, which facilitates having network IDS sensors detecting both known and unknown methods of attack.

Comparing Packet Grepping and Protocol Analysis Signatures

Let’s illustrate the advantage of looking for vulnerabilities instead of exploits with an example. Consider the various exploits and vulnerabilities involving the FTP protocol. If you search the Internet for information on FTP vulnerabilities, you would find dozens, maybe even hundreds of them. Now imagine how many different FTP exploit programs and scripts exist, and how many variations on those programs there could be. And imagine how many of these programs aren’t publicly available. So how can you create a good signature set if you base your signatures on just the exploit programs that you can find? Of course, the answer is that you can’t. You’ll certainly catch some attacks with the signatures, but who knows how many you will miss?

An additional issue is that of timeliness. Obviously, if you are creating a signature set based on known exploits, you can’t write a new signature until you see the exploit. After an exploit has become publicly available, the signature writer must acquire a copy of the exploit it, analyze or test it to determine how it works, and then develop, test and distribute a signature based on the exploit’s characteristics. By definition, this means that a signature that alerts when the exploit occurs will not exist until some time after the exploit is publicly available. In most cases, there will be a significant delay between the time the exploit is first used and the time that the IDS can recognize its activity.

Now let’s think about signature development from the opposite point of view. Rather than focusing on writing signatures to match exploit programs, let’s create a signature set based on known and potential vulnerabilities. A signature set based on protocol analysis will have extensive knowledge of the protocol’s expected behavior. The FTP signature set would be aware of all valid FTP commands and would note any unknown commands that it detected. This could identify various security issues, such as a non-FTP application being run on an FTP port, or a buffer overflow attempt that provides a string of a hundred ‘a’ characters for a command name. Another part of the signature set could verify that particular FTP command arguments contain only alphanumeric characters, and alert on any arguments that contain binary data, such as shellcode.

Other parts of a protocol analysis-based signature set would do additional validations on various commands and arguments, to look for any other signs of abnormal or suspicious activity. These validations would correspond to known vulnerabilities and likely unknown vulnerabilities, such as buffer overflows and illegal values. Of course, we can’t psychically predict what unknown vulnerabilities exist. But we can anticipate them by checking various fields for abnormal values. There are two primary reasons for this:

1. The vulnerability may involve an unexpected value for a particular field. For example, the largest valid argument for command FOO may be 64 characters, but an attacker specifies an argument with 500 characters.
2. The vulnerability may be coded poorly, using unusual values which are noteworthy but unrelated to the vulnerability. An example would be a protocol header value that is nearly always set to a value of 1, yet a particular exploit happens to be coded to set that value to 16.

Attacks in both categories can be identified using protocol analysis; we can catch these attacks by validating the protocol’s data. In the first category, we can create a signature that will identify the likely intent of the attacker’s exploit. In the second category, we may not necessarily be able to determine the purpose of the attack, but we can determine that unusual activity is occurring and likely needs further investigation. So rather than basing a signature set on the characteristics of exploits, we instead base the signature set primarily on a careful analysis of the given protocol.

Performing Protocol Analysis on FTP Traffic

When we perform network intrusion detection using protocol analysis, we are examining the header values and/or payload values for the protocol, as applicable. This can cover an incredible range of signatures, from looking for certain strings in email headers that indicate an email-borne worm or virus, to a suspicious URL that could indicate a web server attack, to a FTP command whose argument contains shellcode. For the sake of illustration, we will look at the characteristics of some real and hypothetical FTP vulnerabilities, and discuss how protocol analysis techniques can identify them.

A buffer overflow vulnerability in the FTP MKD command might be exploitable by providing a very long argument to it containing shellcode. Typical packet grepping signatures contain sequences of the shellcode (often 10 to 20 bytes) and need to match it exactly anywhere within the FTP packet. Protocol analysis allows us to identify the argument to the MKD command and to then verify that it is not overly long and that it does not contain binary data, both of which are conditions we'd expect to find if a buffer overflow exploit is being attempted. By checking this argument "generically" instead of looking for particular shellcode sequences, we will find many different attempts to exploit it, rather than just the few known exploits. Also, attackers cannot easily avoid detection by making a change to the shellcode; the packet grepping signatures would be fooled, but the protocol analysis signatures would not.

The FTP command "SITE EXEC" can be used to execute commands on an FTP server, and it has been used in many attacks. "SITE" is the actual FTP command name, and "EXEC" is an argument to the "SITE" command. Packet grepping signatures look for "SITE EXEC" in a packet, trying to make a case-insensitive match of that string. Protocol analysis takes a different approach, looking for the FTP command "SITE" with the argument "EXEC". So what's the difference? Well, attackers may try to evade detection by IDS sensors by adding superfluous whitespace between "SITE" and "EXEC", such as several extra spaces. Many FTP servers ignore the additional spaces, so they see "SITE EXEC" and "SITE   EXEC" as having the same meaning. Obviously, a packet grepping signature that just does a string comparison will not be able to match these two strings; a protocol analysis signature that understands how to parse "SITE EXEC", or any variation thereof, into its key components will still be able to identify this attack accurately.

Conclusion

In this series, we have been learning the basic concepts of network intrusion detection signatures. This article has introduced the concept of protocol analysis, which means that the network IDS actually understands how various protocols, such as FTP, are supposed to work, and verifies that the traffic it sees follows the expected behavior for that protocol. Protocol analysis has several benefits over more primitive signature techniques, such as packet grepping. By using a protocol analysis-based IDS solution, you will have much better detection of both known and unknown attacks. By focusing on anomalies within the traffic, rather than simply looking for the signatures of particular exploits, protocol analysis-based signatures are much more difficult for attackers to evade through changes to exploits' code or IDS obfuscation techniques.

In part one we discussed the basics of network IDS signatures and then took a closer look at signatures that focus on IP, TCP, UDP and ICMP header values. In the second installment we looked at some signature examples. In the previous article, we began to examine the topic of protocol analysis, which means that the intrusion detection system actually understands how various protocols, such as FTP, are supposed to work. Now, we will continue to look at protocol analysis and how it can overcome attempts by attackers to obfuscate their exploits so that they cannot be detected by simple intrusion detection signature methods.

Protocol Analysis Review

Let's quickly review what protocol analysis means. In protocol analysis, the network intrusion detection system (IDS) sensors examine TCP and UDP payloads, which contain other protocols such as DNS, FTP, HTTP and SMTP. The sensors understand how these protocols are supposed to work, based on RFCs and on real-world implementations of the protocols, and they can fully decode them. This allows a much larger range of signatures to be created than would be possible through simpler signature techniques. Some IDS sensors can only utilize "packet grepping" signatures, which do character-by-character or byte-by-byte matches within a TCP or UDP payload. Although packet grepping signatures are useful for identifying certain types of attacks, they lack the flexibility to identify many types of attacks. In particular, packet grepping signatures are typically unable to handle the obfuscation techniques that attackers often use to attempt to evade detection by IDS sensors.

In the previous article, we reviewed a simple example of how an attacker can attempt to obfuscate an attack through the use of extra white space. If an intrusion detection system uses a packet grepping signature that looks for the FTP "SITE EXEC" request, it will be looking for that exact match in the TCP payloads. However, most FTP servers ignore extra white space, so an attacker could place an extra space between "SITE" and "EXEC", and the packet grepping signature would fail to make a match because of the space. Protocol analysis signatures would break the whole "SITE   EXEC" command into its components, the command name "SITE" and the argument "EXEC", making the extra white space irrelevant. This is probably the simplest way by which attackers obfuscate attacks; now let's dig in and look at some more interesting examples of attempting to evade detection.

Path Obfuscation

Another simple IDS evasion method that many attackers use is path obfuscation. The idea of this technique is to alter the path so it has a different appearance but the same meaning. This technique is most frequently used within URLs to hide HTTP-based attacks. Here are three of the ways that attackers commonly use to obfuscate paths. In these examples, we'll assume that we want to alert whenever we see `scripts/iisadmin` as part of the path in the URL.

Backslashes are substituted for regular slashes. Most Web servers don't care whether backslashes or regular slashes are used to separate directories. So the Web server will treat the URL excerpts `scripts/iisadmin` and `scripts\iisadmin` the same way. Unfortunately, IDS signatures that just do simple text matching will see these two strings as being different and will not generate an alert if `scripts\iisadmin` is used.

Single-dot sequences, like `/./`, may be added to the path. When a single period is listed as a directory, it refers to the current directory. When it's used within a path, it is completely ignored by the Web server. So `scripts/./iisadmin` has exactly the same meaning as `scripts/iisadmin`. Again, a simple packet grepping signature won't identify `scripts/./iisadmin` as an attack.

A slightly more complex technique utilizes double-dot sequences, such as `/../`, to further obfuscate paths. A double-dot directory means that the parent directory should be used. How does this help an attacker? Well, the attacker can list an irrelevant directory immediately before a `/../` sequence; the `/../` will wipe out that directory. So if the attacker uses `scripts/sample/../iisadmin`, the `/../` will wipe out `sample`, leaving `scripts/iisadmin` again.

Of course, an attacker may mix and match these methods. She can use multiple instances of any of these methods within a URL, as well as using multiple methods in a single URL. For example, the attack could be hidden as `scripts/././iisadmin`, or as `scripts\./iisadmin`. Obviously, there are countless obfuscation variations for this URL, so packet grepping signatures can't possibly catch all the attempts. So how can we create a network IDS solution that overcomes these obfuscation techniques?

Protocol analysis is able to handle these techniques because it performs much of the same processing on URLs that a Web server or operating system does. When HTTP traffic is being monitored, the IDS sensor extracts the path from the URL and analyzes it. It looks for backslashes and single-dot and double-dot directories, and it handles them appropriately. After it has "standardized" the URL, it can then search it for likely suspicious directory content, such as `scripts/iisadmin`. Now let's look at a stronger example of how protocol analysis performs the same processing that a server would do: handling hex encoding within a URL.

Hex Encoding

Hex encodings can be used to represent characters in URLs. You may have seen URLs that contain `%20`. This is a hex encoding, which is the equivalent of a single space. Because a URL cannot contain an actual space, it will use `%20` instead if a directory name or filename contains a space. All alphanumeric characters have hex encoding equivalents that can also be used within URLs. Unfortunately, although Web servers understand hex encoding, many intrusion detection signatures do not. This provides a golden opportunity for attackers to utilize hex encoding to evade IDS detection. Fortunately, we can de-obfuscate such attacks through the use of protocol analysis-based signatures.

Let's use our `scripts/iisadmin` example again. An attacker wishes to try to use this directory in a URL but does not want an IDS sensor to detect its usage. So he uses a hex encoding in place of one of the characters. `%73` is the hex encoded equivalent of `s`, so he alters his URL to use `script%73/iisadmin`. Hex encoding gives attackers countless ways to obfuscate each URL that they use; the best way to identify the underlying attack is to rely on protocol analysis. Signatures that perform HTTP protocol analysis perform the same hex decoding that a Web server would do, so they first convert the `%73` to `s` before evaluating the URL for potential attacks. Actually, as you're about to see, Microsoft IIS Web servers perform two rounds of hex decodings, which gives attackers yet another way to obfuscate their attacks.

Double Hex Encoding

In September 2001, the Nimda worm spread throughout the Internet. It took advantage of a vulnerability that was named the Escaped Character Decoding Vulnerability, which involves double hex encoding. An attacker could craft a URL so that it contained special hex-encoded sequences. When a vulnerable Microsoft IIS server received a URL, it performed one round of hex decoding on the path in the URL. It performed a security check on the URL at that point, but afterwards performed a second round of hex decoding on it. Let's look at an example from the Nimda worm to illustrate how this worked.

The Nimda worm used 16 different URLs to probe Microsoft IIS servers for known vulnerabilities, including the double hex encoding one. One example of the relevant part of the URL is `scripts/..%255c../winnt`. `%25` is the hex encoding equivalent of the `%` character. So when the URL is hex decoded, that text becomes `scripts/..%5c../winnt`. `%5c` is the hex encoding equivalent of a backslash. So when a second round of hex decoding is done, `%5c` is hex decoded and the URL will contain `scripts/..\../winnt`. As we learned earlier, this is a path obfuscation of `scripts/../../winnt`, which is a type of attack known as a root traversal. Thorough HTTP protocol analysis will perform two rounds of hex decoding on the entire URL path in order to identify such obfuscation attempts.

Unicode (UTF-8)

The final obfuscation method that we will look at in this article is Unicode (or, more correctly, UTF-8). This is another alternate way of representing characters and is most infamously known for being used for various HTTP-based attacks. When you first look at Unicode, you might think that you are really looking at hex encoding. For example, `%c0%af` is a valid Unicode sequence, not two consecutive hex encodings. Hex encodings correspond to ASCII characters up to %7f; these values are higher, so we can recognize them as Unicode and not hex encodings.

An example of a URL path excerpt containing Unicode is `scripts/..%c0%af../winnt`. `%c0%af` is the Unicode equivalent of a slash, so this path actually means the same as `scripts/../../winnt`, the same root traversal attempt we just looked at in the Double Hex Encoding section. By including Unicode handling in the HTTP protocol analysis signature capabilities, we can convert Unicode sequences to their regular ASCII equivalents and identify the underlying attacks that were obfuscated.

Signature Example

Now that we understand why handling obfuscation is so important, let’s look at an example of how we would make a signature for a particular attack. This signature will be composed not only of the de-obfuscation methods we have already reviewed in this article, but also on the other signature principles we studied in earlier articles in this series. Let’s go through the steps that the IDS sensor and signatures would perform to identify URLs that include `/hidden/admin/` in their paths:

1. Decode the IP packet header to determine what protocol the IP payload contains. In this example, we are looking for IP protocol number 6, which corresponds to TCP.
2. Decode the TCP header to find the TCP destination port number. Assuming that our web server listens on port 80, we would look for that value as the destination port. This tends to indicate that the user is sending an HTTP request to the server.
3. Rely on HTTP protocol analysis to parse the HTTP request into all of its many components, including the URL’s path.
4. Process the URL path by handling path obfuscation, hex encoding, double hex encoding, and Unicode.
5. Review the de-obfuscated path for a match with “/hidden/admin/”, and generate an alert if a match is found.

This is a great example of how intrusion detection, and protocol analysis in particular, is really done. What sounds like a simple signature – looking for `/hidden/admin/` – is a much more complex process than you might initially think. But that level of complexity is necessary because attackers can literally use an infinite number of attack variations. The only way of identifying all such attacks with network intrusion detection is by performing protocol analysis.

Conclusion

In this article, we have continued to examine the topic of protocol analysis, where network IDS sensors understand how protocols such as FTP and HTTP are supposed to work, and decode and analyze them accordingly. Protocol analysis-based signatures provide a superior intrusion detection solution, compared to other signature methods, because they can detect a much wider range of attacks, including known and unknown attacks. As we have examined in detail, protocol analysis-based signatures are also far more resistant to attackers' obfuscation attempts than other signature techniques.

The next article in this series will continue to examine protocol analysis, focusing on the concept of stateful protocol analysis. Stateful protocol analysis is when protocol analysis is performed on an entire protocol session, and key attributes are tracked throughout the session. For example, the IDS sensor can pair a command with its corresponding response, and it can verify that commands are issued in the correct sequence. Stateful protocol analysis is an incredibly powerful signature technique that we will examine in depth in the next article.

This is the fifth and final installment in a series of articles on understanding and developing signatures for network intrusion detection systems. In the previous article, we looked at the topic of protocol analysis, meaning that the intrusion detection system actually understands how various protocols, such as FTP, are supposed to work. We initially looked at protocol analysis as it applied to a single request or response. In this article, we will extend this discussion by looking closely at stateful protocol analysis, which involves performing protocol analysis for an entire connection or session, capturing and storing certain pieces of relevant data seen in the session, and using that data to identify attacks that involve multiple requests and responses.

Stateful Protocol Analysis

The concept of stateful protocol analysis is simple: to add stateful characteristics to regular protocol analysis. When we perform protocol analysis, we examine TCP and UDP payloads, which contain protocols such as DNS, FTP, HTTP and SMTP. IDS sensors that perform protocol analysis understand how each protocol is supposed to work, based on RFCs and on real-world implementations of these protocols. So the IDS sensor can detect many suspicious values within protocol application payloads. Protocol analysis signatures can also be designed to overcome attempts by attackers to obfuscate their exploits. For example, hex encoding can be used to slightly modify URLs so that they appear different to us but have the same meaning to Web servers. Microsoft IIS Web servers perform two hex decoding operations on URLs before processing them: an HTTP protocol analysis signature set looking for IIS attempts should also perform two hex decoding operations, so that it is examining the same URL that the IIS Web server would see.

Although protocol analysis by itself is a very powerful technique, it is limited to examining a single request or response. Of course, many attacks cannot be detected by looking at one request - the attack may involve a series of requests. The best way to detect such attacks is by adding stateful characteristics to protocol analysis. When we perform stateful protocol analysis, we monitor and analyze all of the events within a connection or session. The IDS sensor can "remember" significant events and data for the duration of the session. This allows the sensor to find correlations among different events within a session, identifying attacks with multiple components that cannot be detected otherwise. Without the ability to keep state, we can only examine each packet, request or response on its own, completely independent of the rest of the session.

Pairing Requests and Responses

The concept of stateful protocol analysis is best explained by looking at a few examples that illustrate its power. There are several ways to think of "state" in a connection. One of the simplest forms of state is being able to associate a request with its corresponding response. Because an IDS sensor that performs stateful protocol analysis can monitor every request and response in a session sequentially, it is easily able to match a response with the request that generated it. In many cases, this is extremely valuable, as many responses contain some sort of status indicator that tells us what the result of the request was. For example, when an FTP command is issued, the server's response will start with a three-digit code. A status code that starts with a "2" indicates that the command was successful, while a "5" indicates failure.

IDS sensors can make use of this information in multiple ways:

- The most obvious way is to tell whether certain attacks succeeded or failed. If an attacker attempts to retrieve a sensitive file, and we see that the status code in the response starts with a "2", then the attempt was successful.
- Because we can identify each failed request and keep track of how many requests have failed in a session, we can look for large numbers of failures. This is very useful in finding brute force attacks, such as password guessing attacks.
- A more sophisticated use is to identify instances where a request does not have a matching response. Some attacks involve rapidly sending the same command to a server hundreds or thousands of times. In these cases, the IDS sensor may see many requests from the client before seeing a single response from the server.

As you can see, the simple capability to remember the last command and associate it with its response gives us the ability to identify several types of attacks.

Saving Data

Here's an example of when we would save data from a session for the IDS sensor to use during the rest of the session. An FTP user logs in, using a four-step process:

1. The user sends "USER kkent" to the FTP server, where kkent is her username.
2. The FTP server responds with "331 Send password".
3. The user sends "PASS foobar" to the FTP server.
4. The FTP server confirms that the password is correct and responds with "230 User logged in."

The IDS sensor detects the USER command and stores "kkent" as the tentative username for this FTP session. It then watches for a PASS command that has a response with a status code that begins with a 2, which indicates that the authentication was successful. (The IDS sensor also watches for another USER command, which indicates that the user is attempting to authenticate again.) Once the successful authentication has been detected, the IDS sensor knows that the session it is monitoring is that of username kkent.

This information is useful to us in two main ways. First, if an attack occurs during this session, the IDS can report that username kkent was used for this session. This information may be extremely helpful when investigating an incident. Second, imagine that a more interesting username than kkent was used – perhaps "root" or "administrator" was used. By statefully analyzing all the authentication-related requests and responses, the IDS sensor can detect attempts to use such accounts, plus it can record whether each was successful or not.

Verifying the Sequence of Commands

Another way to think of state is the phases of a session. Let’s look at the phases of an FTP session to gain a better understanding of the benefits of tracking state.

1. Connection: The FTP client establishes a TCP connection to the server’s FTP port (TCP port 21).
2. Authentication: The user of the FTP client either sends a username and password, or an “anonymous” login, to the FTP server. The user remains in this state until either the authentication is successful, or the connection is terminated.
3. Transaction: Once the user has been authenticated, he or she can issue commands to the FTP server. If any of these commands cause data to be transferred, such as files copied between the client and server or directory listings, a separate short-lived TCP connection is established for each transfer.
4. Disconnection: When the FTP client or server wishes to end the session, the TCP connection is torn down.

Being able to identify which state a session is in is the only way of identifying certain kinds of attacks. For example, some FTP attacks and reconnaissance methods rely on issuing transaction-phase commands before the user has successfully authenticated. The only way we can accurately determine when this has occurred is by monitoring the whole session for a successful authentication, so when a vulnerable transaction-phase command is issued, we can generate an alert if a successful authentication hasn’t yet occurred. This is a special case of a larger goal of stateful protocol analysis, verifying the sequence of commands (we will look at that in more depth shortly).

In the FTP transaction stage, an additional TCP connection is generated every time FTP-related data needs to be transferred between the FTP client and server. Before establishing an FTP data connection, the FTP client and server communicate with each other (through commands and responses) about which IP address and port should be used. If the FTP client wants to initiate the data connections to the server (passive FTP), it will send a command to the FTP server, and it expects the FTP server’s response to contain a port number. The FTP client will then attempt to establish a new TCP connection to that port on the server. An IDS sensor that uses stateful protocol analysis can detect the FTP server responses that tell clients what port numbers to use for data connections. When it sees connections involving the FTP server, it can then confirm whether they are legitimate FTP data connections, or whether an attacker is attempting to connection to the server in order to gain unauthorized access to data. This is a great example of how the tracking of state can identify attacks that otherwise could not be detected.

As mentioned earlier, we can utilize stateful protocol analysis to verify the sequence of commands. We’ve already discussed how this can be useful when identifying the phases of a connection, such as making sure that a user has successfully authenticated before issuing certain commands. We can also use state to check for particular sequences of commands. For example, when you rename a file in FTP, two commands are used. First, a RNFR (rename from) command provides the old name of the file; after the server’s positive response to this command is received, a RNTO (rename to) command should be issued. There are attacks that involve issuing many consecutive RNTO commands. Since a RNTO command should be preceded by a RNFR, we should be suspicious if we see ten consecutive RNTO commands. Advanced signatures based on stateful protocol analysis can count how many consecutive RNTO commands are seen, or they can remember the previous command and when a RNTO is issued, check the previous command.

Conclusion

In this article, we have completed our examination of protocol analysis by studying stateful protocol analysis. Stateful protocol analysis performs regular protocol analysis on each request and response in a session, but adds several benefits because it can track various types of state within each session. By adding the ability to pair requests with their corresponding responses, several more types of network intrusion detection signatures can be created to identify attacks that could not otherwise be found. Because we can extract and "remember" important values within a session, we can correlate various events within a session to identify attacks that involve two or more requests and responses. Another benefit of tracking state is that we can identify the phases of a connection and verify that the commands we see take place in the proper phase. Similarly, we can confirm that commands are issued in the proper sequence. Stateful protocol analysis is the best way, and in many cases the only way, to identify various complex attacks.

Karen Kent Frederick (kkent@bigfoot.com) is a senior security engineer for the Rapid Response Team at NFR Security. She is a graduate of the University of Wisconsin-Parkside and is currently completing her master's in computer science, focusing in network security, through the University of Idaho's Engineering Outreach program. Karen has over 10 years of experience in technical support, system administration and information security. She holds several certifications, including SANS GIAC Certified Intrusion Analyst, GIAC Certified Unix Security Administrator, and GIAC Certified Incident Handler. She is one of the authors of "Intrusion Signatures and Analysis", and she is a contributing author to the "Handbook of Computer Crime Investigation".