# 36. Intrusion Detection

In this class, we've talked about many ways to prevent attacks, but not all defenses are perfect, and attacks will often slip through our defenses. How do we detect these attacks when they happen?

Imagine that you're managing a local network of computers (for example, all the web servers and employee computers in a company's office building). The local network is connected to the Internet with a router (recall that all requests from the local network to the wider Internet will pass through this router). How can we detect attacks on this network?

## 36.1. Types of detectors

There are three broad types of detectors. The main difference in implementation is where on the network these detectors are installed. Each type of detector has its advantages and drawbacks.

## 36.2. Types of detectors: Network Intrusion Detection System (NIDS)

A NIDS (network intrusion detection system) is installed between the router and the internal network. This means that all requests to and from the outside Internet must pass through the NIDS. The NIDS can see (and potentially modify) all packets sent to the outside Internet and received from the outside Internet.

The biggest advantage of a NIDS is that a single NIDS is enough to cover the entire network. There's no need to install anything on the end hosts (e.g. employee computers or web servers) because all their requests will pass through the NIDS anyway. Installing a single NIDS for the whole network is a cheap solution with low management overhead.

However, there are some drawbacks to using a NIDS. Recall that even though the Internet fundamentally works by sending packets, rich information communicated with higher-layer protocols are made up of multiple packets. For example, a single message sent through TCP may consist of many small packets that are combined to form a longer message. Also, packets may be dropped or sent out of order–it's the end hosts' responsibility to rearrange the pieces correctly with TCP.

A plain NIDS that just observes individual packets would not be too useful, because it will probably see a lot of packets with partial data out of order. The NIDS may also be seeing packets from lots of different TCP connections, since every connection from inside the network goes through the NIDS. A more useful NIDS would separate packets by their connection and correctly reorder the packets within each connection together by TCP sequence number. Once the NIDS has successfully reconstructed the connection, it can read the rich information and analyze it for attacks.

To make matters worse, the TCP connection reconstructed at the NIDS may not match the TCP connection that the end host sees. Recall that each TCP packet has a time-to-live (TTL) field, which specifies how long the packet can be in transit before it expires. (This is often measured in the number of hops, i.e. the number of machines that the packet has been sent through.) Then there could be a scenario where the NIDS receives a TCP packet because the TTL has not yet expired, but by the time it's sent to the end host, the TTL has expired, and the end host discards the message. There could also be a scenario where the NIDS sees a packet, but it gets corrupted or dropped before it reaches the recipient. Thus the NIDS must also reason about packets that potentially don't reach the end host.

The possibility of inconsistent interpretations of messages between the NIDS and the end host can be exploited for attacks. Consider a NIDS that raises an alert for an attack if it encounters the string `/etc/passwd` in any request. An attacker could send a packet with the content `%65%74%63/%70%61%73%73%77%64`. To a basic NIDS doing string matching, this won't look like an attack, but if the end host is expecting a URL-encoded string and decodes this string, then the end host will receive the string `/etc/passwd`. The NIDS has failed to detect a potential password attack! This type of attack, where the attacker tries to obfuscate the contents of an attack, is called an *evasion attack*. The possibility of evasion attacks suggests that not only does the the NIDS have to reason about inconsistent information about connections, but the NIDS must also reason about how the end hosts may potentially interpret the information in the connection.

Another major issue with NIDS is the need to deal with encrypted traffic. Most modern web traffic is encrypted with HTTPS (TLS), which is end-to-end secure. In other words, the NIDS has no way to determine the contents of the messages being sent. To allow NIDS to analyze encrypted traffic, the network may need to be configured so that the end hosts give the NIDS their private keys to allow the NIDS to decrypt TLS connections. This might not always be a desirable solution, since it compromises the security of private keys and the security guarantees of NIDS, and it may allow network analysts to see sensitive information that only the end hosts should see.

## 36.3. Types of detectors: Host-based Intrusion Detection System (HIDS)

A HIDS (host-based intrusion detection system) is installed directly on the end hosts. For example, antivirus software might be considered a HIDS, because it is installed on the same computer that is generating and receiving network requests.

HIDS have much fewer inconsistency issues than NIDS. Since the HIDS is located on the same machine that is receiving and interpreting the requests, it can directly check what data is received and how the data is being parsed. HTTPS connections are also no longer an issue, because the HIDS can view the decrypted traffic at the end host.

However, these advantages don't come for free. Unlike NIDS, where a single implementation can defend against the entire network, a HIDS must be installed for every machine on the network. This can be very costly, especially if different machines need differently-configured HIDS.

HIDS also don't defend against all evasion attacks. For example, a web server might expect a filename input from the user and serve the matching file to the user. If the user inputs `evanbot.txt`, the server might check the `/public/files` directory and return the `/public/files/evanbot.txt` file to the user. An attacker could supply a malicious input like `../../etc/passwd`. In Unix, `..` says to go up one directory, so this input would allow the attacker to access the passwords file, even though it's located in a different directory on the server. This type of attack is called a *path traversal attack*. To fully defend against path traversal attacks, it is not enough for the the HIDS to understand the contents of the end request. The HIDS would also need to reason about how the underlying filesystem interprets the contents of the end request. This can lead to further parsing inconsistencies and evasion attacks.

## 36.4. Types of detectors: Logging

A third approach to intrusion detection is logging. Most modern web servers generate logs with information such as what web requests have been made, what files have been accessed, and what applications have been run. We can analyze these logs for evidence of malicious behavior or attacks.

Logging is similar to HIDS because both systems directly use information from the end host, avoiding many potential parsing inconsistencies and problems with encrypted traffic. However, like NIDS, logging may need to consider evasion attacks such as the path traversal attack, which requires smarter filesystem parsing and can lead to inconsistencies.

The biggest drawback to logging is that it cannot be done in real-time. By the time the log has been generated, the event that's being logged has already happened. This also means that if an attack has happened, a log-based system will only detect the attack after it has happened. This can be dangerous if the attack is immediately damaging. However, logs are still useful for detecting attacks after they've happened (better late than never).

In terms of cost, logging is usually cheap, because web servers already have built-in logging mechanisms. The only overhead is occasionally running an external script on those logs to search for evidence of attacks.

## 36.5. False Positives and False Negatives

There are two ways a detector can go wrong. A *false negative* occurs when an attack happens but the detector incorrectly reports that there is no attack. A *false positive* occurs when there is no attack, but the detector incorrectly reports that there is an attack. As an example, consider a fire alarm system. A false negative occurs if there is a fire but the fire alarm does not go off. A false positive occurs if there is no fire, but the fire alarm goes off.

It's easy to build a detector with a 0% false negative rate. Just report that there is an attack every single time. Then there will never be a case where your detector incorrectly reports that there is no attack. Similarly, a detector that never reports an attack will have a 0% false positive rate. Clearly, both of these are pretty useless detectors. In the real world, different detectors will have different false negative rates and false positive rates, and part of designing a good detector is balancing the two error rates. In general, as one error rate decreases, the other error rate will increase. Intuitively, to get fewer false positives, you must alert less often, which means you will also incorrectly fail to alert more often (higher false negative rate). Similarly, to get fewer false negatives, you must alert more often, which means you will also incorrectly alert more often (higher false positive rate).

Suppose you have two detectors. Detector A has a false positive rate of 0.1% and a false negative rate of 2%. Meanwhile, Detector B has a false positive rate of 2% and a false negative rate of 0.1%. Which of these detectors is better? It depends on the cost of each type of error. Consider the fire alarm system–if the fire alarm gives you a false negative, then your building has burned down, but if the fire alarm gives you a false positive, then you've wasted an hour with the fire department. In this scenario, the false positive is probably less costly than the false negative, so you would probably prefer Detector B. In another scenario, a false positive might be more costly than a false negative, so you might prefer Detector A instead.

The quality of your detector also depends on the rate of attacks. Consider Detector A again. If we receive 1,000 requests a day and 5 of them are attacks, then the expected number of false positives is $0.1\% \times 995 \approx 1$ request per day. (995 requests are not attacks, and out of the non-attacks, 0.1% of them will incorrectly be reported as an attack.) However, now suppose we receive 10,000,000 (10 million) requests a day and 5 of them are attacks. Now the expected number of false positives is $0.1\% \times 9{,}999{,}995 \approx 10{,}000$ requests per day. Note that nothing has changed about the detector. The only thing that changed was the number of requests received per day (and thus the rate of attacks). However, in the second scenario, our detector is much less useful, because we have to

handle 10,000 false positives every day. This example shows that accurate detection is very challenging when the rate of attacks is extremely low, because even a very good detector will flag so many false positives that it becomes impractical to manually review every single false positive. For more information on this phenomenon, read about the base rate fallacy.

## 36.6. Detection strategies

So far, we've talked about how detectors are installed and how to measure their effectiveness, but we haven't talked about how the detector actually analyzes network traffic to detect an attack. There are four main strategies for detecting an attack, each with their benefits and drawbacks.

## 36.7. Detection strategies: Signature-based detection

**The idea**: Look for activity that matches the structure of a known attack.

Signature-based detection can be thought of as *blacklisting*–we maintain a list of patterns that are not allowed, and we detect if we see something in the list of disallowed patterns.

**Example**: We know that inputting some garbage bytes, followed by a memory address, followed by shellcode is the structure of a buffer overflow attack, so the detector can search for strings that match this pattern and classify them as attacks.

Pros:

- Detecting known signatures is easy.
- It's very good at detecting known attacks. Over time, the security community has built up huge shared libraries of attacks with known signatures.

Cons:

- It won't catch new attacks without known signatures.
- It might not catch variants of known attacks if the variant is different enough that the signature no longer matches. If the signature detector is too simple, it's easy to modify the attack slightly to circumvent the detector.

## 36.8. Detection strategies: Anomaly-based detection

**The idea**: Develop a model of what normal activity looks like. Flag any activity that deviates from normal activity.

Anomaly-based detection can be thought of *whitelisting*–we maintain a list of normal patterns that are allowed, and we detect if we see something that is *not* in the list of allowed patterns.

**Example**: A C program expects user input. Most user input consists of letters, numbers, and symbols–things you would expect a user to type on a keyboard. We determine that normal activity is any input that can be typed on a keyboard, and flag any input that cannot be typed on a keyboard. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that often can't be typed on a keyboard), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.

Cons:

- Defining normal behavior is difficult. What if you train a model for normal behavior on training data that includes attacks?
- A poor model might classify lots of attacks as normal, or classify lots of normal requests as attacks.

In general, anomaly-based behavior is mostly studied in academic papers but not widely deployed as a detection strategy.

## 36.9. Detection strategies: Specification-based detection

**The idea**: Manually specify what normal activity looks like. Flag any activity that deviates from normal activity.

Specification-based detection is also a form of whitelisting. The main difference between specification-based detection and anomaly-based detection is that specification-based detection manually defines normal activity (instead of trying to learn a model for normal activity).

**Example**: A C programmer writes a program that asks for the user's age as input. The programmer knows that ages are numerical and specifies that normal behavior is inputting a number. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that are not numbers), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.

- If the specification is well-defined, the false positive rate can be made very low.

Cons:

- It's very time-consuming to manually write specifications for every application.

## 36.10: Detection strategies: Behavioral detection

**The idea**: Look for evidence of compromise.

Unlike the other three models, behavioral detection doesn't search for attack patterns in the input. Instead, behavior detection looks for malicious behavior that an attacker might try to perform. In other words, we are looking for the result of the exploit, not the contents of the exploit itself.

**Example**: A C programmer writes a program that never calls the `exec` function. If an attacker tries to input a buffer overflow attack with shellcode that calls the `exec` function to spawn a shell, we detect that the code has called `exec` and flag this behavior as an attack. Note that we did not analyze the attacker input. Instead, we analyzed the program behavior and noticed that it called the `exec` function, which is evidence that the program has been compromised.

Pros:

- It can catch new attacks that have never been seen before.
- If the behavior rarely or never occurs in benign (non-attack) circumstances, the false positive rate can be made very low.
- It can be cheap to implement.

Cons:

- The attack is only detected after it's started, so there's no way to prevent the attack before it happens.
- An attacker can try to avoid detection by using different behavior to execute their attack.