Network and Web Security

Answers to Selected Questions for Tutorials 3 and 4*

February 11, 2021

Tutorial 3: Network Security Tools

2 Packet sniffing and analysis with Wireshark

1. Imagine that an attacker were able to use `kali-vm`. What kind of network attacker would they be? (Refer back to the slides for Lecture 7 and think about the capabilities the attacker would have on the `dirtylan` network, based on what you've just seen in Wireshark.)

   With respect to the rest of the `dirtylan` *network, an attacker using* `kali-vm` *could be any of the three network attacker types discussed in Lecture 7. Because promiscuous mode is enabled on* `kali-vm`*'s* `dirtylan` *virtual network adapter, they can passively monitor all traffic originating from and destined for hosts on* `dirtylan`*'s 10.6.66.0/24 subnet (as evidenced by the packet captures that can be performed in Wireshark), giving them eavesdropping powers. They can also inject new packets into the network (a fact that is relied upon in the optional exercise in section 5), giving them off-path attacker powers. If they are prepared to perform a rogue DHCP server attack (described in the answer to the next question), they may also be able to manoeuvre themselves into becoming man-in-the-middle attackers against selected hosts on the* `dirtylan` *network.*

2. Consider the DHCP protocol exchange you witnessed. What could an attacker with your capabilities do to disrupt this protocol? What would be the effects of these disruptions?

   Two common attacks are the DHCP starvation attack *and the* rogue DHCP server attack. *A DHCP starvation attack floods the DHCP server with DHCP Discover/Request packets in an attempt to exhaust the finite supply of IP addresses that the server is able to assign, potentially spoofing the MAC address in each pair of DHCP Discover/Request packets so the server believes that each request is being made by a different client. If this attack is successful, it leads to a denial of service to genuine hosts on the network attempting to use DHCP to configure their networking stack, and they may be unable to use the network unless they can fall back onto a manually-specified networking configuration. A rogue DHCP server attack involves setting up a fake DHCP server and convincing hosts on the network to accept configurations offered by it instead of those offered by the genuine DHCP server; it's usually enough to do this by responding to DHCP Discover requests faster than the genuine server, since most hosts will choose between multiple DHCP offers simply by using the configuration offered by the server that responded first. If this attack is successful, it can potentially lead to a man-in-the-middle attack against the host that accepts the rogue configuration: a DHCP offer usually proposes a default gateway to the client, and if the client uses a default gateway under the control of the attacker, all of the client's communication with hosts outside the local subnet will be forwarded via (and therefore be interceptable by) the attacker.*

*Thanks to Chris Novakovic `c.novakovic@imperial.ac.uk` for preparing this material.

3 Port scanning and host discovery with Nmap

1. How did Nmap determine which TCP ports were open during your port scan in step 2? (Look back at the packets that were sent between `kali-vm` and `listener` — filtering on the `tcp.port` field will make it much easier to work with the captured packets.)

   Assuming that you performed a TCP SYN scan (`-sS`), by filtering the captured packets by each possible destination port (e.g., with the filters `tcp.port == 1`, `tcp.port == 2`, etc.), you'll see that Nmap determines whether a given TCP port is open by attempting to perform the first two legs of the TCP connection handshake. Nmap sends a SYN packet to the remote host on the port. If the remote host responds with a SYN/ACK packet, the port is open, indicating that a service is listening on that port; if the response is a RST packet, the port is closed, and no service is listening on that port. (Nmap also determines whether an intermediate firewall is interfering with traffic between the two hosts: if neither response is received within a reasonable timeframe, Nmap assumes that a firewall is blocking either the outbound SYN packet or the response from the remote host.) This procedure is repeated for each port to be scanned.

2. In step 3, why might it have been necessary to apply a timeout to the UDP scan?

   Unlike TCP, UDP has no concept of a connection: if a packet is sent to a closed UDP port, the remote host's networking stack may reply with an ICMP Destination unreachable *packet but isn't compelled to reply at all, and even if a packet is sent to an open port, the service may decide not to reply for arbitrary reasons. This makes it difficult for Nmap's UDP scan mode (`-sU`) to distinguish open ports from closed ports. Network congestion or intermediate firewalls may also cause Nmap's probe or the remote host's reply to be lost, but this may be indistinguishable from the scenario in which the UDP port is simply closed or the service chose not to reply. As documented in the* Port Scanning Techniques *page in Chapter 15 of the Nmap Reference Guide, Nmap therefore waits for a response for a short time after sending a UDP probe and retransmits further probes if no response is received to be more certain that the lack of a response is not caused by adverse network effects; even if this delay only lasts for a couple of seconds per port, this means that a full 65,535-port UDP scan could take over 50 hours. Because of this, a timeout (or something more sophisticated) should be applied to the UDP scan of* `listener`.

   Since we're performing a port scan on a local subnet with no firewall, we need not worry about adverse network effects, and can tell Nmap not to retry in the event that a probe goes unanswered (e.g., `--max-retries 1`). To complicate matters further, the Port Scanning Techniques *page in Chapter 15 of the Nmap Reference Guide also explains that Nmap rate-limits UDP probes (to avoid flooding the network with huge numbers of packets and making it more likely that remote hosts respond with ICMP* Destination unreachable *packets when closed ports are scanned), causing the scan to take even longer. The low default rate limit can be overridden manually (e.g., `--min-rate 10000`) to speed up the UDP scan further, at the possible cost of a less accurate scan.*

3. In step 5, you may have noticed an odd result when Nmap tried to discover details about the services listening on `listener`'s TCP ports. What caused this odd result?

   Assuming that you performed a scan that included service discovery (e.g. `-sV`), Nmap indicates that it was uncertain about the type of service listening on TCP port 22 by reporting the service as `ssh?` — while processes listening on TCP port 22 are typically SSH servers, the `?` indicates that Nmap sent commands conforming to a number of different protocols to `listener` on port 22, but the responses received did not conform to any protocol known to Nmap, and it was therefore unable to establish what type of service is listening on port 22. Nmap's output then asks you to submit a trace of the communication that just occurred on port 22 to the Nmap developers if you know what type of service is listening, so that it can be added to Nmap's database. (Connecting to `listener` on TCP port 22 using Netcat makes it clear that the listening process is indeed not an SSH server: it's a simple text-based greeting server that I wrote. Please don't report it to the Nmap developers: this isn't something they'll want to add to Nmap's database!)

4. How might an intrusion detection system (or an alert network administrator) notice an attacker performing a port scan on a network? Why might the scans you carried out in steps 5 and 6 be more noticeable?

   Although the volume of network traffic generated by a port scan of a single host is low in comparison to benign network traffic, it usually has a distinct signature: one host attempting to establish TCP and/or UDP connections to another host on a large number of ports (in sequential order, if a range of ports is

selected with the `-p` *option) in a narrow timeframe. Unless the scan is deliberately made stealthier (e.g., using Nmap's* `-T` *option), this makes it easier for an intrusion detection system to detect a port scan taking place. The traffic that occurs on scanned ports is especially suspicious when version detection is being performed (e.g., Nmap sending SSH, HTTP, RTSP and DNS protocol traffic in the same connection to TCP port 22 in step 5). The volume of traffic can increase significantly when an entire subnet is being scanned, and this traffic has its own distinct signature (e.g., ARP requests for each IP address in turn in the subnet), which makes subnet-wide scans more noticeable.*

Tutorial 4: Server-Side Web Vulnerabilities

3 Gathering information on `dvwa`

1. Why do you think Nmap's deduction of `dvwa`'s operating system version is so broad?

   The OS Detection *page in Chapter 15 of the Nmap Reference Guide explains that Nmap detects the remote host's OS version by fingerprinting its networking stack — this is possible because there are subtle differences in the header content and ordering of packets created by different OSes and different versions of the same OS. Nmap has a database that maps known networking stack fingerprints to OS versions. It determines that* `dvwa` *is running a version of the Linux kernel between versions 3.2 and 4.6; this range is so broad because the networking stacks in these kernel versions are similar enough that they cannot be distinguished simply by inspecting the header content and ordering of packets they create. Notably, this fingerprinting method is also unable to distinguish between kernels provided by different Linux distributions.*

2. Can you find a more cunning way to deduce `dvwa`'s operating system version which is (probably much) more accurate than Nmap's method? (Hint: Nmap provides you with the information, but isn't doing anything with it…)

   Even though minor OS versions and Linux distributions may not be distinguishable by fingerprinting their networking stacks, additional information is often leaked by services at the application layer; Nmap's OS detection functionality doesn't currently take this information into account when deducing a remote host's OS version. In `dvwa`*'s case, the banner of the web server listening on TCP port 80 (*`Apache httpd 2.4.10 ((Debian) PHP/5.6.29-0+deb8u1)`*) reveals that the listening process is version 2.4.10 of the Apache web server, with version 5.6.29 of the PHP interpreter being used to execute PHP scripts in the web server environment. The PHP version string itself leaks even more information:* `deb8u1` *is a suffix prepended to packages compiled for Debian 8, so it's reasonable to assume that this is the Linux distribution that* `dvwa` *is running (and, indeed, it is).*

   We could go even further and use this information to take a guess at which version of the Linux kernel is running on `dvwa`*. There's a good chance that* `dvwa` *is running a Linux kernel available in the Debian 8 repositories. From the Debian Package Tracker's entry for the* `php5` *package, we can see that PHP 5.6.29 was added to the Debian repositories on December 17th, 2016 and removed on February 18th, 2017; cross-referencing these dates with those in the Debian Package Tracker's entry for the* `linux` *package, we see that Linux kernel versions 3.16.36 and 3.16.39 were the only ones present in the Debian 8 repositories in this date range. Assuming that the administrator of* `dvwa` *installed all package updates around the same time, it's therefore likely that* `dvwa` *is running one of these two kernel versions. (*`dvwa` *is indeed running Linux 3.16.39.)*

3. How could the administrator of `dvwa` make it more difficult for an attacker to discover the information you just found?

   There's no reason that the Apache web server listening on TCP port 80 needs to disclose so much version information in its banner. The easiest way to limit this leakage is to configure Apache to display the minimal amount of information possible in its banner, which can be done using the `ServerTokens` *directive in the Apache configuration file.*

5 Finding vulnerabilities in DVWA

1. How could the command injection vulnerabilities in DVWA be fixed?

   The value of `$target` *in the PHP code should be properly sanitised before it is passed to the* `shell_exec()` *function. Removing known shell metacharacters using the* `str_replace()` *function is not ideal, since mistakes can be made this way (as in the high security level, where the pipe character is only removed if followed by a space). Ideally, since the intention is for* `$target` *to contain an IP address, the PHP code should ensure that the value does indeed look like an IP address; this can be done using PHP's built-in* `filter_var()` *function, although DVWA's impossible security level implements its own validation routine.*

2. How could the file upload vulnerabilities in DVWA be fixed?

   The PHP code expects an image to be uploaded, but should not rely on client-provided information about the file (i.e., the MIME type in `$_FILES['uploaded']['type']`*) when deciding whether this is the case. More robust checks can be performed to test whether the content of the file is in fact an image, although it's possible to construct a file that appears to be an image while also containing code that can be executed by the PHP interpreter, so this isn't entirely foolproof. The best way to fix this vulnerability is to disallow the client from controlling the name of the file that is written to disk (a hash of the file contents could be used as the file name instead), and to configure the web server so that files stored in the upload directory cannot be executed.*