CS 161 Computer Security
Spring 2010 Paxson/Wagner
Discussion 5

1. **DNS** Recall that in a blind DNS spoofing attack, the attacker tries to guess the identification number of the DNS request sent by the victim.

   (a) In a blind spoofing attack, the attacker must know when a DNS request is about to be made by the victim so that he can respond with his attack responses. How might the attacker know when the victim is about to make a DNS request?

   (b) What can an attacker do if he successfully gets a victim to believe his bogus DNS mapping?

   (c) How can an attacker avoid having to carry out this attack for every request made by the victim?

   Answer:

   (a) Lure a victim to your web site, which contains a link (or many links) to the site whose DNS record you would like to attack (e.g., `google.com`). When you see that a victim has contacted your site, you know they are about to make a DNS request for `google.com`, so you initiate the attack at this point.

   (b) He is in control of the content of any request made to a hostname whose DNS record has been successfully spoofed. For example, if the attacker managed to get the victim to accept a bogus DNS record for `google.com`, then any subsequent request to `google.com` will actually go to a domain of the attacker's choosing. He might get you to reveal your password to the fake `google.com` at that point.

   (c) DNS records are cached, so the attacker might set a long TTL (time-to-live) so that the bogus DNS record will live in the cache for a long time. The attack will succeed for as long as the bogus DNS record lives in the cache.

2. **Firewall Misconfigurations** Suppose you had a ruleset that looked like this:

   1 drop tcp 10.1.1.0/25:* *:*

   2 allow udp *:* 192.168.1.0/24:*

   3 drop tcp 10.1.1.128/25:* *:*

   4 drop udp 172.16.1.0/24:* 192.168.1.0/24:*

   5 allow tcp 10.1.1.0/24:* *:*

   6 drop udp 10.1.1.0/24:* 192.168.0.0/16:*

   7 allow udp 172.16.1.0/24:* *:*

   Which rules contradict one another? Can you find more than one example?

   Hint: (Hint: Look for when all the packets one rule intends to deny (accept) gets accepted (denied) by a preceding rule?)

Answer: Rules 2 and 4. All packets to 192.168.1.0/24 are accepted by rule 2, but rules 4 seeks to deny the subset of packets from 172.16.1.0/24 to 192.168.1.0/24. This is called shadowing.

Rule 5 is shadowed by the combination of rules 1 and 3. Rule 1 says deny any IP that matches the first 24 bits 10.1.1, followed by a single 0 bit. Rule 3 says deny any IP that matches the first 24 bits 10.1.1, followed by a 1 bit. Combined, this is equivalent to saying deny any IP that matches 10.1.1, which is what rule 5 states.

3. **DoS** It was recently discovered that a large broadband provider in the United States was attacking end users' BitTorrent connections with a layer-4 Denial of Service attack. For each BitTorrent connection, they would inject a single packet into the network, which would tear down the connection.

   (a) How might this have worked?

   (b) Does TCP defend against this attack? Explain how it does or why it doesn't.

   (c) How could a different attacker (i.e., not the ISP) launch this attack? Assume that the attacker is another home broadband user.

   Answer:

   (a) The ISP can send a single RST packet to tear down a TCP connection, as specified by the protocol.

   (b) No, because the ISP is on the path between the two endpoints of the connection. The ISP can simply look at the TCP sequence number and inject a packet with the right sequence number. They do not need to guess a sequence number for this attack to succeed. The same is true of any man-in-the-middle.

   (c) This would be harder. They would have to guess the TCP sequence number in order to inject a RST packet that would be accepted.

4. **Reference monitors** We learned in class that firewalls are an instance of the more general concept of a reference monitor.

   (a) What is a reference monitor? What are the 3 properties of a reference monitor?

   (b) We would like the following to be reference monitors. Decide whether each of the following items actually is an example of a reference monitor and explain why or why not. If not, which of the 3 properties do not hold?

      i. The Calnet login page
      ii. Your OS controls access to the files on your local computer
      iii. Your web browser's Same Origin Policy

   Answer:

   (a) A reference monitor is an access control mechanism that satisfies three properties: (1) complete mediation (always invoked), (2) tamper-proof, and (3) verifiably correct.

   (b) None of them really qualify because they are not verifiably correct.

      i. Controls access to student-specific functions, e.g., viewing grades, adding/dropping classes, etc. Always invoked, or at least we hope it is. May or may not be tamper-proof, but certainly intended to be tamper-proof. Probably not simple enough to be verifiably correct.
      ii. Doesn't ensure complete mediation; can just boot another OS to get to all the files. Not tamper-proof; again, could boot another OS and modify the working of the OS access control mechanism. Definitely not simple enough to be verifiably correct.
      iii. Ensures complete mediation (although XSS is a separate issue) — every web site is subject to it, and it cannot be turned off. Tamper-proof absent any browser vulnerabilities. Probably not simple enough to be verified.

5. **Attacks on NIDS** Network intrusion detection systems (NIDS) examine the content of packets entering and leaving the network to detect specific attacks and take an appropriate action, such as dropping a packet or logging a warning. Snort is a well known example of such a system. Typically, NIDS allow you to define a series of rules specifying the action to be taken when a packet matches the "signature" of some known malicious content. For example, one rule may trigger an alert if it detects the transmission of a Microsoft Word document containing a known exploit.

   The simplest signatures are simply strings which are matched verbatim against the payload of the packet. However, it is often easy to evade these by slightly modifying the content of an attack, so some NIDS allow more sophisticated signatures written as regular expressions or algorithms. Below is an example of a Snort rule one might write to detect certain HTML content that suggests an attempted XSS attack:

```
alert tcp any any -> 192.168.1.0/24 80 (pcre:"<script/src.*?=.*?>.*?</script>")
```

   In the above, each ".*?" sequence just matches any sequence of zero or more characters, and will try to consume the minimal number of characters possible while ensuring the entire signature matches (if it can).

   (a) What problems might be caused by matching against more sophisticated signatures if someone can send an arbitrarily long set of bytes?

   (b) Why is this so bad? How can an attacker take advantage of this?

Hint: Regular expressions as we know them are usually "Perl Extended Regular Expressions." Those types of regular expressions cannot be encapsulated by a finite automaton. Consider backtracking in regular expressions.

Answer:

(a) If the signature matching rules are known ahead of time by an attacker (as they often are because they are based on public rule sets), an attacker can construct sets of bytes that cause "worst-case" matching by the NIDS. If, for example, the NIDS has to backtrack in a regular expression continuously, this will dramatically decrease the speed at which a NIDS matches signatures.

This is known as an "algorithmic complexity" attack.

The regular expression (pcre = perl compatible regular expression) in the example Snort rule could suffer from this problem, especially if it had more ".*?" sequences. In this case, the worst case input would contain all the given literal strings except the last (the closing script tag). When such an input is matched, the final closing script tag will fail to match, causing the regex engine to backtrack, giving up the match for the ">" character before it, and allowing the preceding ".*?" to consume more characters in the hope of finding a second ">" character followed by a closing script tag. After that fails, the engine will backtrack to the preceding ".*?" sequence and then read all the way to the end of the input again. The end result of this process is a matching time something like $O(n^k)$ (I think ...), where n is the input length and k is the number of ".*?" sequences.

(b) If a NIDS is slowed down significantly, there are a few ways to handle it. One way is for the NIDS to deal with packets and not let packets through until all packets are inspected. However, if the NIDS is slowing down by a factor of a million or so, this likely will add unacceptable performance overhead to your network.

Another option, and the more common solution, is if the NIDS is too busy processing, either drop packets to deal with new ones or just deal with old packets slowly. This means that either you drop potential attacks, or you let threats through and slowly process other attacks, potentially not discovering attacks until a much later point.

Thus, it comes down to a choice of either the NIDS missing attacks or slowing down your network to unacceptable rates. This is effectively a Denial of Service attack on the NIDS, rather than the server.