Exercise Set 4

helge206	Andrew Helgeson
wayt0012	Ethan Waytas
maure113	Brian Maurer

1.

SRC ADDR	DEST ADDR	SRC PORT	DEST PORT	PROTOCOL	ACTION
10.1.100.100	*	*	sendmail	TCP	ALLOW
*	10.1.100.100	*	sendmail	TCP	ALLOW
*	10.1.100.200	*	shh	TCP	ALLOW
10.1.200.200	*	*	http/https	TCP	ALLOW
*local_subnet	0.1.2.3	*	ssh	TCP	ALLOW
*local_subnet	42.42.42.42	*	ssh	TCP	ALLOW
*local_subnet	3.14.15.9	*	ssh	TCP	ALLOW

The last 3 rules could be made more generic by having the *SRC ADDR* be *. This does not expose any hosts on the intranet since the destination addresses are external to the subnet. So if an external packet (to the company) arrived with destination address 0.1.2.3 and destination port 22 it would not go to any computer in the local network.

Potential Vulnerabilities

- Any of the allowed connections could still be malicious. Just because they are allowed by our firewall doesn't mean they are safe. This cannot be mitigated by the proxy or firewall. The services running on the open ports must be trusted.
- There is still a vulnerability to IP fragmentation attacks such as overlapping fragments. If the network stack on end hosts are not careful this can be a security risk. This wouldn't be stopped by the firewall but for web traffic the proxy could prevent it. Since the proxy would operate on the application level HTTP protocol it would reassemble packets before forwarding them on to the final host. At this point the proxy could do the fragment checks and not leave it up to the potentially vulnerable end host.
- The ability to ssh to uncontrolled external hosts (the last 3 rules) would allow someone on the local subnet to tunnel un-approved internet traffic through these hosts. This could be any type of traffic assuming these hosts don't have any specific rules on their network. The local proxy and firewall would not be able to prevent this security breach without

denying ssh access to these external hosts. If the external hosts are also controlled by the company, the firewall for their local network could be setup to enforce the same rules but there is no mention of this.

- This set of rules does not prevent a DoS attack on any of the exposed services within the network. Even if the routes to these services are closed removing *allow* rules, an attacker can still achieve saturation of the incoming network link, preventing the firewall from processing valid traffic.
- A host on the local network could spoof the source address of an HTTP packet to be coming from 10.1.200.200, the proxy server. This would then be allowed through the firewall and potentially to a non-approved site. On the return however, the TCP packet will be addressed to the proxy server which will not recognize the TCP connection and drop the packet.

2.

(a) ***An old Snort rule says that any HTTP packet that includes "/..%c0%af../" should trigger an alarm, as an attempted IIS exploit. Explain why in "normal" usage this rule would have a low false positive rate.***

The false positive rate for an IDS is defined as $\frac{\#\ False\ Positives}{\#\ Normal\ Events}$

On a busy network such as a school or business there would be a large amount of HTTP traffic. The attack being detected by this Snort rule attempts to traverse up the directory of the web server, potentially accessing files the requester shouldn't. It is highly unlikely that this particular sequence of characters that expands to "/../../" would be part of a valid HTTP packet. Valid HTTP traffic would almost always just use the absolute path to the resource. Since this is a highly unlikely pattern to appear in normal traffic you can be fairly confident that any alert based on this rule is valid, even in the presence of high volumes of traffic.

(b) ***Suppose Eve discovers a web server, vulnerable.org that is vulnerable to the IIS unicode exploit and she wants to exploit the hole without having it noticed. What are a few ways Eve can temporarily increase the false positive rate at vulnerable.org for the rule, without getting her IP address noticed?***

To trigger a false positive, Eve will have to request a resource from vulnerable.org that contains the attack signature but is not actually trying to access unauthorized content (i.e. outside of the web site). Here is such a url: vulnerable.org/dir1/dir2/..%c0af../dir1/dir2/cat.gif (equivalent to vulnerable.org/dir1/dir2/cat.gif)

This url will request the image cat.gif but first it will go to dir2, go back to the web root and then back to dir2. This is a valid directory to request on this server but according to the Snort rule above, this would be flagged as an attack. The other challenge to overcome is to make many request for this resource without having her IP address flagged/banned which would stop her from making the actual attack. There are several ways she could do this:

- Eve could use a botnet to flood requests to vulernable.org. This would provide significant traffic/noise and most request would be coming from different addresses so stopping the flood would be difficult.
- Eve could post the image on a popular site like reddit (they love cats...) so that every time someone loads the page they also request the image from vulnerable.org.
- Eve could also use a VPN or proxy. This likely wouldn't be the best choice because the proxy/vpn address could be banned just as if Eve was doing it herself.

Regardless of which method she uses to spam the false attack requests, Eve's end goal is to flood some human admin who has to check these alarms so that her actual attack will likely go unnoticed.

(c) ***What can you conclude about "advertised" false positive and false negative rates?***

You could have very low false positive and false negative rates and still be vulnerable. This relates to the base rate fallacy. In the presence of large amounts of traffic, even a very low false positive rate can create an overwhelming number of alarms. The simple example given in class showed that on a network with 10M flows per day, 100 of which are intrusions and a 0.1% FPR you would still end up with 10,000 false alarms. So to truly judge the usefulness of an IDS system you need to put its false positive and false negative rates into context with the amount of traffic the system handles.

More related to false negative rates is the effect of a knowledgeable or sophisticated attacker. An attacker who has knowledge of the system and how it detects intrusions can likely find a way to circumvent it, thus increasing your false negative rate. If the attacker is knowledgeable or just doesn't have the time to try and circumvent they could use a flooding technique like Eve did above to overwhelm the system with false positives so the real attack can proceed hopefully unnoticed.

The measurement of the false negative rate for an IDS system is kind of after-the-fact since if you know what wasn't caught you would have had a rule for it in the first place. So in a way our false negative rate is only measurable to the extent of our current knowledge of the attack and possible circumventions. These are things that will always be changing so in my mind the false negative rate for a rule based IDS system like this isn't that useful.

3.
(a) ***Does ViruSniff violate the undecidability of the halting problem? Why or why not? (Hint: is there a simple program that can do exactly what Sam says ViruSniff can do?)***

The question statement does not say anything about what happens for non-virus executables. ViruSniff could simply yell "VIRUS!!" at every executable. This program would be "100% effective" at detecting viruses but its false positive rate would be so high that it would be useless.

If you interpret the “100% effective” statement to mean that it is completely effective at detecting all and only the true viruses (ie 0% false positive) then yes, it does violate the undecidability of the halting problem. Here is a simple proof of it’s undecidability:

To show that ViruSniff, or more generally the idea of perfect virus detection, is undecidable we will use Rice’s theorem. Rice’s theorem states two properties that must be met in order for a language (a program) to be undecidable:

> Let S be a language of programs with perfect virus detection as ViruSniff claims to have.
> If S satisfies the following properties:
> - For all programs P_1 and P_2, where $L(P_1) = L(P_2)$, $<P_1> \in S$ if and only if $<P_2> \in S$
> - There exist programs $<P_{in}> \in S$ and $<P_{out}> \notin S$
>
> Then S is undecidable

Let P_1= ViruSniff and P_2= ViruSniffImproved, where ViruSniffImproved is the same code as ViruSniff, but also adds a for-loop before print out “VIRUS!!!” that runs 10 times and does nothing (the purpose is to create a different executable than ViruSniff so that $P_1 \neq P_2$).
It is trivial to see that $L(P_1) = L(P_2)$, as the only difference between the two programs is a for-loop that does nothing to change the output. Therefore, $<P_1> \in S$ if and only if $<P_2> \in S$.

Let P_{in}= ViruSniff and P_{out}= Reject, where program Reject always rejects on any input. From above, we know $<P_{in}> \in S$. Clearly, if P_{out} rejects everything, it will never find a virus, so $<P_{out}> \notin S$.

Therefore, by Rice’s Theorem, it is undecidable to figure out if a program is a virus (and therefore ViruSniff violates the undecidability of the halting problem).

(b) ***Explain how to change any program that runs for at least 10001 instructions, and does not trigger the VIRUS!!! alert, to propagate a virus such that the altered program will also fail to trigger the alert. What does your strategy say about Sam’s claim?***

Given a non-virus program (according to ViruSniff) with 10001 instructions you could alter it by inserting a call to your virus code at the 10,0001st instruction. Assuming the previous last instruction (the 10,001st) was the final return from the executable, it would now come after the virus code. Since ViruSniff only checks the first 10k instructions of the program’s execution, there will be no visible change to the program in the eyes of ViruSniff so if it wasn’t detected before, it won’t now. For any longer programs you would do the same thing. Place the call to the virus code right before the last instruction.

To do the above modification the attacker must be able to determine where in the executable is the 10Kth instruction is reached. It isn't as simple as adding the code right before the return from main since this may not be the path that takes 10K+ instructions (also have to consider varying input). The attacker would have to emulate execution of the program much like ViruSniff does to find the 10K instruction mark. The code can be inserted after this but care must be taken with loops. If the code is blindly placed within a loop it may be executed before the 10K instruction mark.

This strategy says that Sam's claim is complete bs. It is really only effective as long as the execution of the virus comes within the first 10k instructions of the executable. Even then, depending on how the "fancy signature matching" works there could be a way to circumvent ViruSniff while still executing the virus code within the first 10k instructions. An attacker could also insert a loop that is equal to the instruction execution amount of ViruSniff to make it give up and then run the virus code.

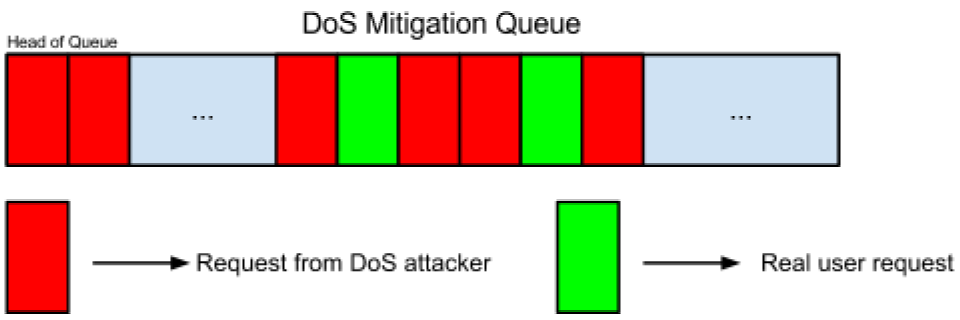
(c) ***Assuming that ViruSniff has some false negatives, how would you go about changing ViruSniff to detect a virus that uses your strategy? What effect would this change have on false positives?***

Our method above simply inserts a call to the virus code at the very end of the program, outside of the 10k instruction check of ViruSniff. One potential change would be to increase the number of instructions we check, but this won't detect all programs altered with our method. If you increase the instruction count to 20k then there will be a slightly longer program that wouldn't be detected. As mentioned previously, an attacker could simply insert a loop that would chew up however many instructions ViruSniff checks and then break out and call the virus code.
A potential strategy for determining if a certain program is a virus would be to analyze the entire executable for a virus signature. If a program being analyzed matched a known signature to some percentage, that program would be declared a virus. The false positive rate would increase if the signatures were not specific enough to a virus in which case they would flag many normal programs. The false positive rate would decrease if the signatures were very specific, although the false negative rate would likely increase as some virus' would be missed due to the narrow scope of the signatures.

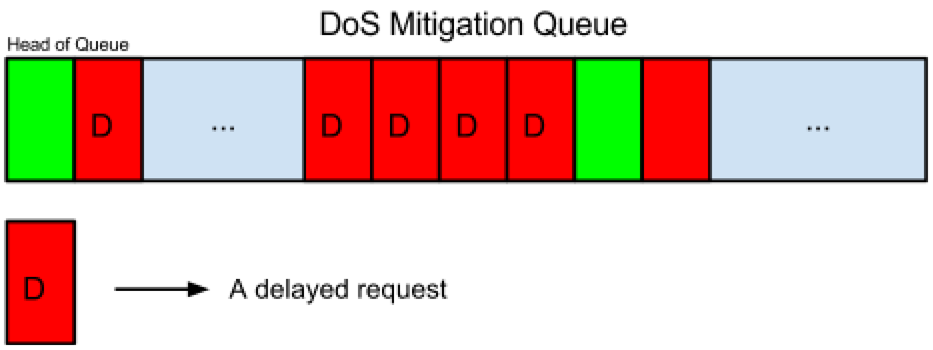
Another method for modifying ViruSniff would be integrity checking using hashes. This method assumes that ViruSniff, or some other program, would first hash some or all of the files on the filesystem. The hash signatures for these programs would be stored somewhere on the filesystem which already presents a vulnerability since viruses could corrupt the stored "valid" hash for the program they are infecting. At some point later, ViruSniff would run a scan of the filesystem and report a virus for any program whose hash has changed. This would detect programs altered using our method above but it would also cause a lot of false positives when system files or other programs receive legitimate updates and their contents change.

4.
(a) ***Will Sly's scheme work, or not, and why? Give a detailed explanation.***

No, Sly's scheme will not work. The queuing of HTTP requests may delay the attack for a little while but once the imposed delay runs out on requests they will be processed at the same speed they initially came in. Here is an illustration of Sly's queue at the start of a DoS attack.



Initially the queue is filled with many phony requests that are part of the DoS attack and only a few valid user requests. In Sly's scheme, assuming the DoS packets are coming from the same address, most of the red packets above will be delayed and put at the end of the queue as shown below.



As seen here the initial DoS requests, with the exception of the first which may get processed right away, are now coming back to the front of the queue after being delayed. For now assume that they are reaching the head of the queue more than 1 second after they initially arrived. This means that when they reach the front of the queue they will start to be processed by the server at the same rate that they initially came in. Meanwhile the attacker can keep stacking up more requests.

While we have shown that it isn't necessary to circumvent this delay to carry out the attack, there are some other methods the attacker could take to while completely avoiding the delay. One would be to utilize a large botnet to spam traffic at the site. This would result in a flood of requests from many different addresses and Sly's delay scheme would change nothing. Even if

Sly doesn't try to fill up the HTTP queue he really only needs to saturate the link coming into the server. This doesn't rely at all on the structure of the delay queue, all Sly needs to do is have more bandwidth than Sly's server, something more easily achieved using the botnet.

The only thing this delay does is postpone the DoS flood for one second. If the attacker can maintain their DoS rate for one second then after that the server will try to start processing them at the rate they originally came in and hence the DoS is no different. The description of this system says it is queueing on the level of HTTP requests not TCP connections so this delay wouldn't cause any more of a resource bottleneck on the attacker's side since one TCP connection could be used to send all the DoS data.

(b) ***Will Carl's scheme work? Why or why not?***

No, Carl's scheme will not work. First of all, many web clients won't have an ActiveX enabled browser (Internet Explorer) so they won't be able to act as a potential server. There is also the concern from the client's perspective that their resources (cpu and network) are now being made available and known to someone else. Unless there is a clear notification and acceptance from the client I would not add them to a p2p like network.

Despite these problems, the HTTP redirect mechanism still isn't a solution. A DoS attack on the primary web server will still severely limit or prevent other users from getting the redirect response, making the DoS effective. Even if some valid clients receive the redirect, they will be going to potentially stale copies of the site stored on other hosts or no hosts may have the page you need if it is a rarely accessed page or dynamically generated. The problem is that while you do have a distributed system for serving up your pages, the knowledge of where to find these is still centralized and vulnerable to attack. BitTorrent uses additional systems such as distributed hash tables and peer exchange to help mitigate the result of a central tracker failure.

Despite the lack of DoS mitigation there are security concerns with having other hosts serve pages. An attacker could set themselves up as one of these mini ActiveX servers and start delivering malicious content to users who are redirected to them. As mentioned above, dynamic pages could not be served by these clients unless you provided them the source code that generated the page. While you shouldn't depend on security through obscurity, you still wouldn't like other people to have the source for your webapp. These dynamic pages typically load content from other systems such as databases which these ActiveX mini-servers would not have access to.

So in short, Carl's suggested solution has problems with compatibility (browser), user permission, centralized failure point and trust in any and all hosts to run a server.