Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 29
November 9, 1999

Problem Set 6 Solutions

Problem 6-1. The Real World

Since there were several relatively good solutions presented, but none that I thought were spectacular, I decided to throw out the "extra point for being the solution" for this problem. Here I've shown some solutions I kind of liked (code cut due to length).

Predictable ISNs (Andy Berkheimer, Raffi Krikorian, Luka Matkins, Gong Ke Shen)

One potential method of exploiting security vulnerabilities for fun and profit are attacks based on exploiting the predictability of initial sequence numbers in TCP. This makes IP spoofing possible, even in situations where the attacking machine is not able to see the traffic between the victim machine and the spoofed machine. Once you can spoof a host, a number of attacks become possible, such as denial of service via connection sniping to excuting commands by violating the trust relationship between the spoofed machine and the victim machine. In this description, we assume the reader already has a working knowledge of TCP and its use of sequence numbers for ordering packets.

How is this done?

There are two cases to be considered here - when the attacking machine has the ability to listen to communications between the victim and spoofed hosts, and when it does not have this ability.

If we can listen in on these communications, then ip spoofing becomes easy to take advantage of, since we can actually see the packets between the two machines. Then we can perform a denial of service attack on the spoofed machine and hijack the connection to the victim machine whenever it meets our fancy to do so. Another less insidious possibility would be to perform a denial of service by sniping the connection immediately any time that we see the two hosts begin the TCP handshake.

If we cannot listen in on these communications, then IP spoofing becomes more difficult to take advantage of, since we cannot see what our victim is sending. Connection sniping or hijacking become nearly impossible, no more likely to suceed than random chance. A more subtle attack can be performed by taking advantage of the fact that many operating system TCP stacks use a predictable algorithm for generating TCP initial sequence numbers.

Predictable ISNs and IP Spoofing

The attack works as follows, given that we know that the victim machine's (V's) ISNs can be predicted in a certain way (specific real world examples of this are given later). First, we find V, and look for other machines which it trusts to log in and execute commands. After finding one such machine (T), we send some sort of denial of service attack (spoofed from a non-existant IP address), or find some other way to take T off of the Internet for a while. If this is not done, then V will send packets to T, and T will terminate the TCP connection.

Then we try and initiate a connection to V, spoofing our IP address as that of the trusted host. Since we're able to predict the ISN that the victim will send out in it's SYN-ACK packet, we can send the appropriate ACK response to complete the connection, and then continue sending data to do whatever we want on the victim. Also, since the victim will usually ignore ACKs with bad sequence numbers, then we can send more than one ACK if we need to in order to guess at one out of a reasonable number of possible ISNs.

Real Life Examples of Predictable ISNs

An interesting study is to look at the many real life cases of predictable ISN generators which have been in extended use on the Internet.

The original TCP specification (RFC 793), suggested that the ISN be incremented by 1 every 4 milliseconds. However, if you can guess what time the SYN-ACK will be sent out at to within a second, then only 250 guesses are needed, which isn't a totally unreasonable number.

One widely used generator is from earlier versions of BSD. This TCP implementation would increment its initial sequence number by 128000 every second, and by 64000 for each new connection made. So by making a few valid connections to the server to see what ISNs it is sending out, it would be easy to guess a correct ISN for a spoofed connection made in the near future.

Windows NT 4.0 has also been found to have predictable sequence numbers. In Service Pack 3, the ISN was incremented by one every millisecond, a small variation of the ISN incrementor from the original TCP specification. They attempted to fix this in Service Pack 4, but it appeared to be even easier to predict, by incrementing the ISN by one of a few possible values every time a new connection was attempted. The next release, Service Pack 5, reverted to the SP3 behavior, and is still at large on the Internet today.

A problem was also found recently in the Linux 2.2.x kernel, where the ISN was computed involving a fast one-way hash of the network addresses and ports of the two machines involved in the connection. It turned out that this function generated similar results when all of the fields except for the source IP address were the same. This could be used to predict ISNs by sending a valid packet from the attacking host to see the current ISN, and then sending a spoofed packet, with the same destination and source ports as the initial "probe" packets.

Given this information, combined with an OS fingerprinting tool such as `queso`, creating an program to perform these attacks would be pretty trivial from the technical perspective (but much more complicated from an ethical one).

Prevention

In a perfect world, everyone would use cryptographic authentication and these attacks wouldn't be possible. Still, it's worth recomending that if you don't already do so, there's no better time than the present to go download an ssh client.

Given that this isn't a perfect world, it i sstill very helpful to use an algorithm such as the one given by Bellovin in RFC 1948[1] in order to generate much harder to predict sequence numbers. This works by combining the counter incremented every few milliseconds with a one-way hash function of the addresses of the machines involved in the communication, along with some other data.

Source of further information

Discoveries of recent instances of predictable ISNs can often be found on the bugtraq mailing list, the archives of which are located at `http://www.securityfocus.com/`.

Another useful source of information for this article were a few articles from Phrack magazine, specifically the articles on IP-Spoofing Demystified[2], and Project Hades - TCP Weaknesses[3].

Lastly, there's no better place to find anything you want to know about the Internet than in RFCs. For this attack, there was useful information in RFC 793 (Transmission Control Protocol specification) and RFC 1948 (Defending Against Sequence Number Attacks).

[1] `ftp://ftp.isi.edu/in-notes/rfc1948.txt`
[2] `http://www.phrack.com/search.phtml?view&article=p48-14`
[3] `http://www.phrack.com/search.phtml?view&article=p49-7`

Password Authentication (Grant Emery, Brian Perrin, Emil Sit)

In this answer we address the question of sniffing and dictionary attacks. These problems are related to protecting the user's password — in Unix, passwords are generally the primary method. Passwords (and pass phrases) are generally preferable to other mechanisms involving mathematics since user-chosen pass phrases are much easier to remember than large random numbers. However, the vulnerability is that once a password is compromised, an attacker will be able to login to the machine and mount any of the local attacks to gain control of the machine.

A typical password authentication will be as follows:

1. The user initiates a service request, identifying himself.
2. The server prompts for a password.
3. The user sends his password to the server.
4. The server verifies this password (or some derivative thereof), usually by comparison against some database stored on the server.

There are a number of possible places where this can be compromised. In lecture 12, two possibilities are described. The first possibility is sniffing of network traffic. This is a concern for many people who are logging in remotely. The second possibility is if a copy of the password database is obtained from a server. An attacker can then mount an off-line dictionary attack against the password database in an attempt to obtain passwords. We focus on the problem of network sniffing.

Sniffers

Password sniffers revolve around the fact that most networking mediums are broadcast based. This means that a host can be configured to process all of the traffic of a network it is attached to. A single compromised machine can run a sniffer which monitors certain connections (such as those for well-known services such as POP, telnet, ftp or rlogin) and parse them looking for the initial authentication exchange. Since these protocols are well-documented, tools can be written to parse most of these exchanges and directly extract a username-password pair. A good example of such a tool is Sniffit.[4] This highly configurable program can monitor your local network traffic at any number of levels of detail, from logging individual packets exchanged by machines, to logs of entire connections, to lists of username-password pairs from telnet, ftp, and pop mail sessions. It also features an interactive mode which allows you to monitor connection status in real time. The package seems to be very well written, and claims it will work under most major Unix variants (Linux, SunOS, Solaris, IRIX, and FreeBSD are listed in its documentation). Additionally, there's even a partner software available for Linux which allows the user to snipe connections.

Another common sniffer is linsniffer which is a sniffer for Linux, as the name implies. This is a much more lightweight piece of code (the version we have[5] is approximately 240 lines of C) which basically just logs all the data flowing into a machine on the telnet, ftp, pop, imap, and rlogin ports. These protocols are the ones which typically contain user passwords in plaintext. This tool has very little in the way of configurability, but it does its job very well. It is frequently seen included in Linux root kits because of its small size and high reliability. For reference, we've included a sanitized portion of a linsniffer log (all usernames and passwords have been changed to protect user privacy) at the end of this discussion. Just through this handful of connections, the program recorded a large number of valid machine logins and email account passwords. It's also interesting to note the difference between the telnet connection from `minoru.mit.edu` to `sorrow.mit.edu` and the connection from `atlantic.mit.edu` to `all-night-tool.mit.edu`. The former was protected by Kerberos, while the latter has a username and password evident.

There are a plethora of other tools beyond these two. In general packet sniffers are incredibly easy to write, so there are a suffusion of them. Linsniffer and sniffit are two of the more common ones found in Linux root kits and installed on compromised machines. Raarpd is one that's commonly found installed on compromised IRIX machines. More sophisticated attacks include installing a sniffing kernel module or, if physical access to the network is available, a specialized sniffing machine could be placed on the network to monitor traffic.

[4] `http://reptile.rug.ac.be/~coder/sniffit/sniffit.html`
[5] `http://www.rootshell.com/archive-j457nxiqi3gq59dv/199707/linsniffer.c`

Prevention

The primary mechanism for dealing with this problem is to deploy protocols which employ end-to-end encryption. This renders the data seen by the sniffer unintelligable. There are two basic ways to do end-to-end encryption. One is at the network layer. This is the method which is employed by IPSec. IPSec and other Virtual Private Network (VPN) products provide encryption and authentication of *all* IP packets. This is particularly appealing because no user applications need to be modified to be aware of encryption. This method of dealing with sniffing has not yet been widely deployed. However, there are many products which are beginning to provide this service. For example, Free S/WAN[6] provides a good IPSec implementation for Linux.

The other option is, of course, to deploy encryption at the application layer. Probably the most popular tool for dealing with this is the Secure Shell (SSH) developed originally by Tatu Ylönen. This primarily deals with the problem of unencrypted logins. This program negotiates a session encryption key generated at random and transferred prior to password exchange using RSA. SSH also does provide support for forwarding other protocols over the encrypted link.

Kerberos is another mechanism for dealing with sniffing. Instead of transmitting passwords, Kerberos uses tickets. These are authentication tokens which are issued by a central Kerberos server (the KDC) and then presented to specific services when needed. To get a ticket, a user simply requests it from the KDC. The KDC will send the user a ticket, specific to the user's machine and the desired service, encrypted using a key derived from the user's passphrase. The user then enters the passphrase locally. Authentication is confirmed upon successful decryption of the ticket. (Kerberos also provides a key in the ticket which the two communicating parties can then use to establish end-to-end encryption.) Unfortunately, Kerberos does not prevent dictionary attacks. A sniffer can record the ticket as it is transmitted and then attempt to decrypt it successfully.

A newer protocol developed at Stanford is called the Secure Remote Password protocol. The security of this protocol is based on the DLP and is at least as secure as Diffie-Hellman. This protocol is verifier based but does not involve transmitting anything that can be used to dictionary attack the password. Implementations and details about SRP are available at `http://srp.stanford.edu/srp/`.[7]

At the network architecture level, it's also possible to cut down on the usefulness of network sniffing by using a switched network architecture instead of the more common broadcast architecture. The difference between a switched network and a broadcast network is that where the broadcast network has hubs and repeaters which copy every incoming packet to every outgoing port, the switched network has packet switches. These are something of a cross between a repeater and a router; they learn which port a specific machine is on, so traffic to that machine is only sent to that port. Thus, packets are only sent to their intended recipients. This is solution helps protect local traffic when encryption is unavailable or there are recalcitrant users who are unwilling to use tools that provide encrypted sessions. It's also good for protecting HTTP traffic and other protocols in which a cryptographic solution is not widely available.[8]

Thus, there are numerous ways of dealing with the problem of packet sniffing. Many of these are even fairly transparent to the user. With a bit of user education and administrator initiative, password sniffing can be made essentially useless.

[6] `http://www.xs4all.nl/~freeswan/`

[7] Site was down at last check.

[8] Although things like SSL are available for cryptographically protecting HTTP transactions, the vast majority of web pages are not available through secure HTTP servers, and encrypting every transfer puts an enormous load on a web server

Fragmentation attacks (Daniel Barkalow, Laura Dean, Nickolai Zeldovich)

We chose to discuss and demonstrate a way to bypass certain IP and TCP packet filters using IP fragmentation, which was briefly mentioned in the lecture on practical UNIX security. The way IP fragmentation works is quite simple in theory: if an IP packet cannot be sent as a whole, the payload of the packet is split into a number of smaller packets, which are then given new IP headers, and sent on their way to the destination. However, this creates certain complications for people who try to filter certain packets in an attempt to keep random people from accessing their systems.

A common way to prevent TCP connections from external systems to internal machines is to place a packet filter on an intermediate gateway which drops all packets coming into the local network with only the SYN flag set. The SYN flag is used to signal a connection request, and if the packet is dropped, seemingly one cannot establish connections from the outside to machines on the local network. But maybe not. If we fragment our initial packet into two packets, the first packet might not contain the SYN flag, and the packet will be passed on by the gateway. The second packet would contain the SYN flag but the gateway already routed the first packet and cannot piece them together in order to match a rejection. So our connection request gets through to the remote machine in two packets, where it is reassembled and processed normally. We are connected!

The way to protect against such attacks is to enable "defragmentation" at the intermediate gateway which is filtering packets. If the gateway reassembles all fragments before passing them on, it will notice that the packet it has reassembled is a SYN request, and will drop it. However, this is prohibitive for other reasons (CPU and memory requirements, possible overflows if you send lots of unfinished segments, etc), and not always implemented.

Here is a demonstration of the exploit we have just described. The "source" machine in this case is *seven-up.mit.edu* (18.239.0.70), and it will try to fake a connection from *moo.mit.edu* (18.239.2.202) to a remote mail server, *orbit.zepa.net* (205.245.53.14). However, the default route on *seven-up* points to a filtering gateway, *geo-prism.mit.edu* (18.239.2.130). The rule set on *geo-prism*, in *ipf* syntax, is as follows:

```
block out proto tcp from 18.239.0.70/32 to any flags S/FSRPAU
block out proto tcp from 18.239.2.202/32 to any flags S/FSRPAU
```

Such that it will try to deny any outgoing connections from either *seven-up* or *moo*, by dropping any packet which has the SYN bit set high. This is typically required to establish a connection, so standard TCP implementations will be prevented from making connections:

```
SEVEN-UP~/class/6.857/ps6> telnet orbit.zepa.net 25
Trying 205.245.53.14...
^C
SEVEN-UP~/class/6.857/ps6>
```

However, we have implemented a simple netcat-like program which will send packets in two fragments, the first being 8 octets (the multiple in which fragment sizes can occur in IPv4), and the rest being everything except the first 8 octets. In the case of the first packet with the SYN bit set high, the first fragment will not contain the SYN flag. (The SYN bit is in the 14th octet of the TCP header, offset 109 bits from the start of the header). The second fragment will contain that information, but since it's a fragment, the filtering code will not bother to check it. And in fact, it works:

```
SEVEN-UP/class/6.857/ps6> ./frag-tcp moo 1234 orbit.zepa.net 25
Using device le0 (0x12EF0000:0xFFFF0000)
<<< SYN+ACK
>>> ACK
220 orbit.zepa.net ESMTP
helo seven-up.mit.edu
```

```
250 orbit.zepa.net
mail from:<kolya@mit.edu>
250 ok
...
```

Using this method, we managed to send a complete email message and successfully close the TCP connection. The code used in this example is available in /mit/kolya/class/6.857/ps6[9]. We used *libpcap* and the *DLPI* interface under *Solaris* to capture and send our TCP packets, respectively. We didn't implement fully RFC 791 and RFC 793 compliant IP and TCP stacks, but just enough to allow connecting to a remote machine via TCP under normal conditions and send/receive data.

[9] Main sections included; supplementary functions omitted due to code length

Problem 6-2. Discrete Logs (Grant Emery, Brian Perrin, Emil Sit)

Given p and q primes, g a generator of both Z_p^* and Z_q^*. and x relatively prime to p and q, the assumptions

$$\begin{aligned} g^a &\equiv x \pmod p \\ g^b &\equiv x \pmod q \end{aligned}$$

are not sufficient to guarantee that the existence of an integer c such that

$$g^c \equiv x \; mod \; n$$

Consider the case of $g = 3$, $p = 5$, $q = 7$, and $x = 2$. We know that $a = 3$ and $b = 2$, but there is no c such that $3^c \equiv x \; mod \; n$.

In order to find the additional conditions necessary for c to exist, we look at how to compute c. We know that any generator raised to the $p - 1$ power mod p is 1 so we can say, for α an integer,

$$g^{a+\alpha(p-1)} \equiv x \; mod \; p$$

Similarly, for q using β

$$g^{b+\beta(q-1)} \equiv x \; mod \; q$$

Also, we know that

$$x \; mod \; p \equiv x \; mod \; (pq)$$

so we can write

$$x \equiv g^{a+\alpha(p-1)} \bmod p$$

$$x \equiv g^{b+\beta(q-1)} \bmod q$$

$$x \equiv g^c \bmod n$$

If c exists, then we must be able to find some α such that $c = a + \alpha(p-1)$ and some β such that $c = b + \beta(q-1)$. To do that, we can set the two equations equal to each other and solve

$$\alpha(p-1) - \beta(q-1) = b - a$$

for either α or β. However, we know that we can do this only if $\gcd(\mathrm{p}-1, \mathrm{q}-1)$ divides $(b - a)$. This is our condition on the existence of c. The gcd of $p - 1$ and $q - 1$ must divide the difference in the exponents a and b. If this condition is met, we can find an α, and our c is $a + \alpha(p-1)$.

Problem 6-3. Common Modulus Systems (Daniel Barkalow, Laura Dean, Nickolai Zeldovich)

The situation is that Alice and Bob have two RSA key pairs between them, having picked a shared (and secret) pair of primes p and q and calculated $n = pq$. Alice's keys are $\{n, d_a\}$ and e_a such that $d_a e_a \equiv 1 \ (\ mod\ (p-1)(q-1))$. Bob's are $\{n, d_b\}$ and e_b such that $d_b e_b \equiv 1 \ (\ mod\ (p-1)(q-1))$, where e_a and e_b are different, so they can tell the difference between their keys.

It happens that e_a and e_b are relatively prime. This lets an attacker recover a message if it has been encrypted for both Alice and Bob.

The attacker finds out n, e_a, and e_b from the public keys. Since e_a and e_b are relatively prime, she can find k_a and k_b such that $e_a k_a + e_b k_b = \gcd(e_a, e_b) = 1$. One of these will be positive and one negative. Without loss of generality, let k_a be the negative one.

When she gets $C_a \equiv M^{e_a} \ (\ mod\ n)$ and $C_b \equiv M^{e_b} \ (\ mod\ n)$, she tries to compute $C_a^{-1} \ (\ mod\ n)$ with the extended Euclidian algorithm. If she fails, $\gcd(C_a, n) \neq 1$, so either $C_a = M = 0$, in which case the message is obvious, or C_a is a non-trivial multiple of a factor of n. In the latter case, she's very lucky, and can find p and q and break the private keys.

Assuming neither of these happens, she can find $C_a^{k_a} C_b^{k_b} \ (\ mod\ n) = M^{e_a k_a + e_b k_b} \ (\ mod\ n) = M$.

Any common factors of e_a and e_b will defeat this particular attack, since they will only give the attacker $M^{\gcd(e_a, e_b)} \ (\ mod\ n)$, which is likely no more helpful than $M^e \ (\ mod\ n)$.