Multiple Levels of De-synchronization and other concerns with testing a IDS system

by Greg Hoglund and Jon Gary

last updated Friday, August 11, 2000

Introduction

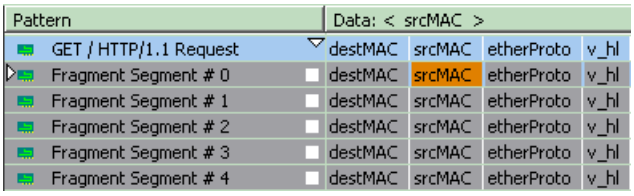
This article is about breaking things. We discuss testing methodologies for your network. Even if you test your own network, be rest assured that someone is doing it for you. Every minute of every day malicious programs are pitted against commercial network applications. While this war rages on, throv a few battalions of idiotic users - not malicious, but certainly not doing what you expect either.

Regardless of what you think, IDS systems are not perfect, nor will they ever be. Don't get us wrong, better to have an IDS system than to not have one - it lowers your risk. But, as an IS Engineer or a Software Engineer you should be aware of the limitations of any IDS system you build or deploy.

Furthermore, you should be aware of the limitations in your network application(s). Your "network application" is all of the software that collectively runs your network services - and you'd be surprised some of the places where it can break. Not just denial of service attacks, but also buffer overflows an bad reactions to unexpected input. Software testing is the only clear solution - without awareness of t weak points, how can you possibly mitigate risk?

"Bad IDS, no doughnut"

Here in the lab, we test a lot of software. A good example is network-based IDS. In the lab, we have b software to test and exploit a number of IDS de-synch problems. We have twisted packets into knots thick that no IDS on the planet could possibly hack through them. History has shown that IDS System can be 'de-synchronized'. In short, 'de-synchronized' means the IDS stops working. The IDS fails to se the attack. A few years ago, a commonly used technique was to fragment your packet streams. This worked well, but today most IDS software can deal with this simple obfuscation.



Our lab software is building IP fragments from a GET request

After an easy to use kernel module became available a few years ago - attackers have been able to automatically 'de-synchronize' all of their TCP/IP traffic. Not until it was a widespread phenomenon dic IDS vendors scramble to fix their products.

Testing and validation is very important - and this doesn't mean running a set of *known* signatures pa IDS and seeing if it fires. We will discuss several basic methodologies for testing *any* network system, just IDS.

Where IDS and Content-Filters will fail

Using IDS as an example, in the best case, a company could purchase one of every commercial IDS they would have the time to correlate the responses from all of them. Of course - they might have to d with more false positives, increasing the time investment. Or, they could outsource this mundane task third-party monitoring company.

There are many solutions deployed for Intrusion Detection, from simple log watching to elaborate hos based syscall analyzers. The largest industry seems to be network-based signature sniffers - and the are popular because they are so simple to deploy - no host-based issues, no integration - it just plugs a promiscuous port and starts singing along. It's a 'point solution' - which means low cost of installatio sounds good to our boss when we tell them we have deployed the latest and greatest IDS - it has goo CYA factor. But - there is a looming cloud on the horizon for both Network IDS and Firewalls.

The fact is that more and more traffic is encrypted every day. IDS systems are going to be hard press to keep up - more processing to decrypt traffic - no longer a point solution because the IDS needs to u keys to decrypt traffic. The IDS becomes part of your infrastructure at that point - no longer just 'an ex bolt in the rack'.

Firewalls are effected by a slightly different phenomenon - the fact that software vendors have simply it. Application developers hate firewalls. The solution? Move all of your application traffic to port 80. L at Microsoft: RPC/DCOM represents a very important direction for Microsoft - so much in fact that Microsoft has now made DCOM available over an HTTP interface. Firewalls are daily becoming more more useless as application vendors account for port filtering by simply moving their services onto po - a port that surely will work over most firewalls.

Firewalls that filter ports are simply rendered useless, leaving the firewall vendors in the position of providing 'content-based' security - in other words - filtering application level requests. Vendors in this space have a huge obstacle to overcome if their filtering and intrusion detection is based on signature The crux of the problem is that content-filters have all the same problems that network IDS systems h had in the past - they can be de-synchronized. This is a classic problem which started with virus dete and has never rested since.

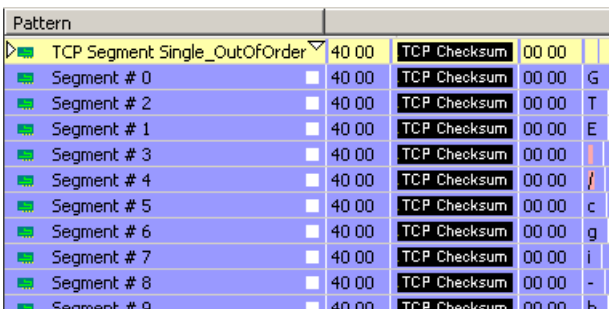
History has already shown us that IDS systems fall prey to a complexity problem - that is, TCP/IP is HARD. It's hard to follow, it's hard to reconstruct - and most of all - it's hard to predict. TCP/IP is on everything - and vendors are dumping all sorts of proprietary stacks on their devices and software. Al means is more chance that a network-based IDS can be de-synced from it's target.

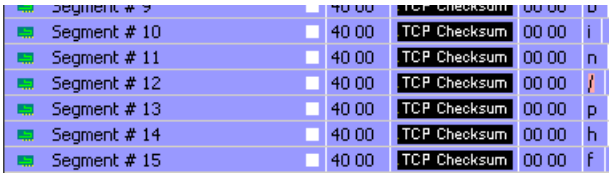
The Testing Strategy

Idea #1: Two Systems can never be exact copies of one another.

'De-synching' an IDS system is an old idea - first introduced many years ago. For many years hackers security professionals have been aware of this potential problem with network IDS. It goes something this--> Attacker inserts extra characters into a transaction - characters which are invalid and will not b processed by the target. However, due to the difficulties in reconstructing the TCP/IP data - the netwo IDS accepts the characters and therefore fails to see what is really going on.

What does it mean to insert or inject into an IDS? How about commands that indicate the reset of a connection? Or, characters that obfuscate the actual transaction - converting an uber-elite 'GET /cgi-l into something more like a 'GET /cgi-bleatin' request (you think I'm kidding don't you?). Additional data cause the header of a packet to appear corrupt, or render an application layer message impossible to decode.





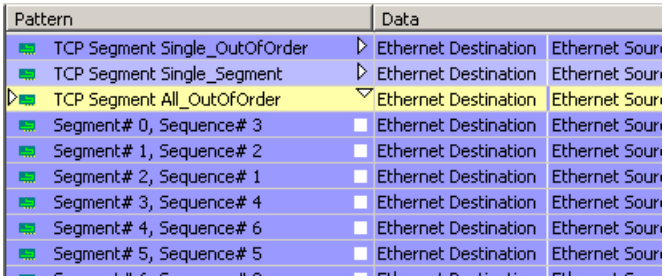
Here we send 'GET' in the form of 'GTE'. This works.

In the same vein, the attacker can cause the IDS system to miss packets that are valid and processed normally by the target. The IDS doesn't fire an alarm because the IDS simply didn't get the packet. Th most obvious example of this sort of attack is the leverage of IP Fragmentation and Overlapping TCP Segments. A few years ago, this method was highly effective against network IDS systems. Even toda these old school techniques can be effective.

Now stop and think for a minute - the stranger the packets that we send, the more an IDS system has work to reconstruct them. For every exception - the 'perfect' IDS has to make the assumption that a pa could have been dropped by the target. The IDS must maintain two states for every packet - accepted and dropped. For every packet, this doubles the number of states that the IDS must follow. For every duplicate or replay packet, the IDS must account for 1/2 again the current load. For an IDS of this natu to even watch for a 10 character signature would require a great deal of memory. Assuming single by packets, the IDS would need 10K of memory just to watch this one session. If my signature is 11 bytes long, the figure doubles to 20K - exponentially increasing in size for each byte thereafter! This is compounded by the fact that the smaller the packets are, the more possible permutations of state. As halve the size of each packet, we double the required size of the IDS buffers. Clearly this is impossibl considering both that the signatures are most likely larger than 10 characters, and that there are close 80 web hits a seconds going over our network. What's the point? There is no such thing as a 'perfect' IDS.

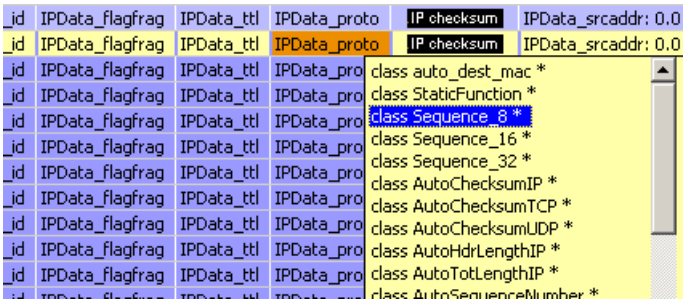
Idea #2: Bury it deep enough and no one will find the treasure.

Take all of these facts one step further. Get crazy and start embedding multiple layers of de-synchronization within one another! For example, a malicious individual could send a malformed cgi POST request in an attempt to get `/etc/passwd` or `\WINNT\repair\sam._`. In any request, these filename strings should set off an IDS. On top of this, use a combination of TCP segment overlap, out order TCP segments, and IP fragmentation/out-of-order fragmentation; the odds are significantly high that the IDS will fail. The IDS must reconstruct the packet exactly as the target machine does. This mi be easy if only one method of de-synchronization were used. Combined, we have increased the likelih that the IDS will fall on its face. It simply doesn't have enough CPU or Memory to keep up with it all.

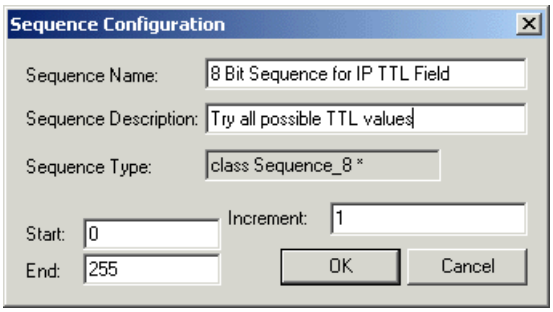


Our lab software is sending TCP segments out of order

We have designed patterns which exercise a variety of TCP and IP headers. A good place to start is t examine the source code of your favorite TCP/IP stack and note all the points where a packet might b dropped or cause a connection to reset. In IP Fragments, we can play with checksums and options. Depending on the target, packets may be dropped if the packet is source routed. IP Options can be invalidated, set to wrong lengths or values. In some networks, the IP fragment can be sent larger than MRU - and we can selectively set the DF (Dont Fragment) bit to cause some packets to be dropped.



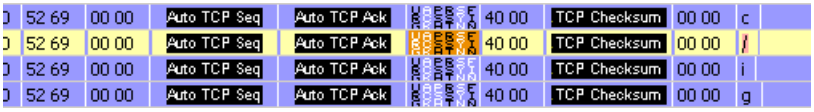
Replacing certain IP header fields with procedural functions



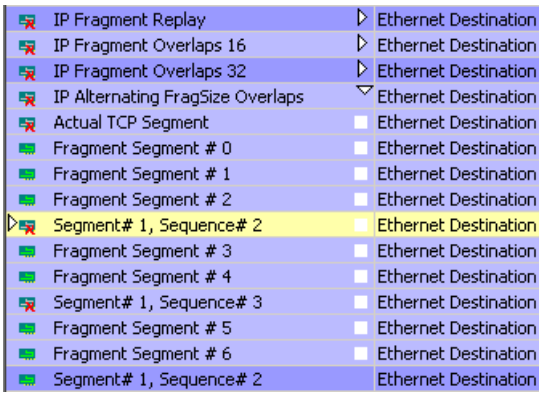
Configuring a number sequence for the IP TTL field

We can also monkey around with TCP Flag options. In at least one test, we simply cycle through ever possible TCP flag combination - including the reserved bits. We can find out if the system is validating sequence numbers on reset packets, for instance. Flag tests alone can generate thousands of packe when combined with other permutations.

We can combine invalid and valid overlapping IP fragments with encapsulated invalid and valid TCP segments. Within these, we can encode malformed cgi requests. *On top of this*, we can replay valid p of the TCP stream. In some cases, the target will accept the replay, and in some cases it will be dropp By alerting the TTL in these packets, some of them might never reach the target system, but easily pa by the IDS system. Remember, we are running all of these obfuscations *at the same time* over the *sa stream*. Even assuming that the IDS can reconstruct a valid request from all of this, we have created a significant load on the CPU.



Invalidating some flags on out-of-order TCP segments for the string `cgi-bin`



Overlapping IP Fragments with Overlapping TCP segments

It would be fair at this point to mention that a bunch of small IP fragments ripping around network is, b itself, enough to warrant an IDS alarm. Although some of these obfuscations work wonders, they also strange. And, if your an attacker, looking strange is the last thing you want. So, some IDS systems wil

alert you when strange constructions of traffic are noticed. However, not *all* obfuscations of this natur are noticed, especially when we start talking about application layer requests and TCP segments.

To make matters worse, protocols layered above TCP/IP have their own sequencing issues. In the ca of SMB/RPC, there are many factors, including UID/MID/TID/FID values - an IDS wishing to correctly asses file access and other windows-networking activity must take-on another protocol. It must accou for a whole new book of invalid possibilities. And, given a protocol like SMB, which has even more exceptions to the rule than TCP/IP - on top of being closed-source, will leave IDS vendors in the dust. Why is this important? An auditing system can sniff all SMB TRANSACT requests in an attempt to see who is looking at what on the network. But, spicing this up with some invalid Session packets might ve well make the auditing system 'forget' what sessions are important.

Another layer of de-synchronization can be added by using HTTPS. For the IDS to reconstruct a pack sent of HTTPS, it has to have the session key. It is unlikely that the IDS will have this key, but even if does, a significant amount of CPU time would be used in the process. In addition, all of the segmentin and fragmentation de-synchronization methods could then be applied to the encrypted HTTPS packe so that if the IDS did not reconstruct the request _perfectly_ it would not even be able to decrypt the request. The vast number of eCommerce web sites using HTTPS for secure transactions run a large number of CGI, ASP, and CFM scripts that are not being monitored by the IDS systems they use. One good example of this is the recent Openhack.com hacking event sponsored by eWEEK. The IDS logs were full of reams of attack attempts, port scans, and the like; but the first compromise that actually worked? Nope, the winning exploit did not even appear in the IDS log. *Why?* Because it was an explo a CGI program running over HTTPS.

Conclusion: Multiple layers of de-synchronization, including additional protocols, application layer obfuscations, and encryption, will be the downfall of network-based intrusion detection.

Idea #3: Cross reference the output with the input (i.e., log files).

De-synch applies to host based IDS as well. When people think of de-synching an IDS system, they usually think of TCP/IP problems and network based IDS. De-synch can also occur on host-based IDS especially those that watch log files. Log file entries are created based on an interpretation - therefore possibility exists that the interpretation is wrong. And, remember that network based ID systems make reports using log files also - so they are not immune from log file issues.

We set-up a traffic pattern to perform a few thousand GET requests to the IIS server - each one slight different. For each request, we compare the resulting log-file entry to the original request. We also cro check the filesystem request against the original request. Discrepancies soon arise.

For example, in the lab we have discovered that the Windows 2000 IIS server makes the following blunder:

```
"GET /+++test.html"
```

is logged as:

```
"GET /+++test.html"
```

The filesystem access is for:

```
"/WEBROOT/+++test.html"
```

Also,

```
"GET /%20%20%20test.html"
```

is logged as:

```
"GET /+++test.html"
```

The filesystem access is for:

```
 "/WEBROOT/   test.html"
```

We have identified a problem - the log file is reporting the WRONG filename in one of our requests. T requests are for two separate and distinct files on the system, yet they are logged the same. The pote exists that someone could covertly hide data on a website and access it using 'malformed' requests. Network IDS systems may falter on the interpretation as well, and host-based log watchers will certair see the wrong data.

Conclusion: By cross referencing the output (i.e., log files) with what we input, we discover inconsistencies.

Idea #4: Look for 'stripped' data - additional data that doesn't effect the validity of a request

Again, by using a special pattern, we generated a variety of requests using special characters and relative paths. It became quickly apparent that the Win2K IIS server was stripping out huge sections c requests. This is very interesting - so we tested against a Redhat/Apache server also. The results we the same for both web servers.

```
"GET /dir1/dir2/../dir2/../dir2/../dir2/../dir2/../dir2/test.html"
'Redundant traversing' is a valid request

"GET /dir1/dir2/this_directory_is_invalid/../test.html"
This request has an invalid path, yet it still works
```

Breaking a request into fields and replacing paths with equivalent paths

Using our traffic patterns we quickly discovered that the relative path trick only works one level deep, is to say "GET /dir1/dir2/this_directory_is_invalid/so_is_this_one/../../test.html" doesn't work.

The potential exists here that an IDS system may not identify the correct request because what it see the wire is not the same as what the filesystem sees on the target.

By the same token, the following pattern also results in a valid request:

```
"GET /dir1/dir2/////////////////////////////////////////////////////test.h
```

The slash characters are extraneous to the request.

Idea #5: Use invalid filename characters

Making requests with invalid characters can reveal bugs in cgi-programs and content-filters - strange characters always present the possibility of mangling parsers. We ran several patterns in the lab that stuffed invalid characters down the pipe and analyzed the results.

The following characters are invalid as part of a filename, yet are passed directly to filesystem calls u Win2k:

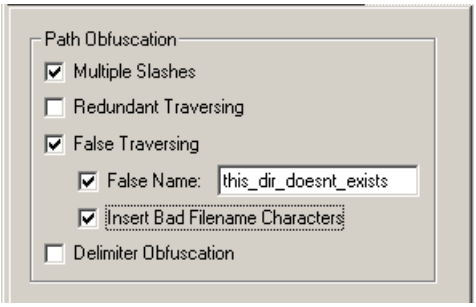
```
\/:*"<>|
```

By simply permuting all combinations of these characters in GET requests, we were able to determine that many of them can be used in a valid request.

These are valid web requests:

```
GET "/dir1/dir2/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\/../test.html"
GET "/dir1/dir2/:|:|:|:|:|:|:|:|:|/../test.html"
GET "/dir1/<><><<>><><||""""""""""""""""""/../test.html"
```

What this means is that a variety of characters, known to be invalid in filenames, can still be inserted i a GET request without effecting the validity of the request.



We built a convenient plug-in to our testing software to simply obfuscate all GET requests

The only filename character that does not work appears to be the '*' asterisk. However, when making request using an asterisk, we noticed a huge spike in processor and file activity. Suddenly we had fou a potential denial-of-service problem. Yet another outcome of simple software testing.

Conclusion: Even invalid characters can be used, if they are filtered out before they hit the target subsystem (i.e., the filesystem).

Idea #6. Exploiting NTFS case sensitivity

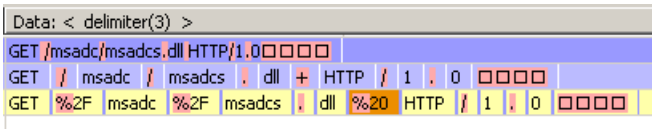
"test.txt" and "TEST.txt" can both be created on an NTFS volume (however, not by the shell). One wa do this is to create the files on a UNIX Samba server and copy them to the NT system. This brings up problem, because IIS always accesses files as all capitals. However, the shell will always return the fi one found (which is usually the first one created). So, 'notepadtest.txt' or 'notepad TEST.txt' will both return the same file, but it may not be the same file that IIS is serving.

Conclusion: By promoting the assumption that the filesystem is case-insensitive, Windows NT has induced an atmosphere where security mistakes can be made. Did your IDS drink the blue coolade to

Idea #7: Use Alternate Character Encoding or Equivalent Characters

Make application-layer requests using a variety of encoding methods. For HTTP, an obvious example to encode characters using their %nn equivalents. "../" can be encoded as %2E%2E%2F - another la of parsing intelligence that a content-filter or IDS would need to recognize. The problem gets worse w you consider that a variety of character encoding methods are available to GET and POST requests. you think about it - your CGI programs are the most vulnerable points in your web site. The mechanis which accept user-input are always vulnerable to exploitation. A POST request, for instance, can be encoded in a number of character sets. Device-in-the-middle parsers have a complexity problem simil that of the TCP/IP issues of old. The device in the middle is not the target parser - and potentially can over characters.

Breaking a request into fields and replacing certain characters with equivalent characters



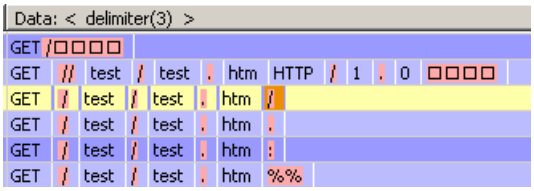
More equivalent characters

Changing what you use as a delimiter character can be interesting. A delimiter is very important and a IDS parser will likely depend on the delimiter. If an equivalent delimiter character can be inserted inste of the default, the IDS or Proxy may fail. Ideas that come to mind are ',' -vs- ';' and '" -vs- ""'. Using a sir pattern you can create many thousands of combinations of these until an acceptable delimiter is foun

Conclusion: Equivalent requests can be made using entirely different character streams. Does your follow?

Idea #8: Add/Remove *trailing* meta-characters

The task of designing these attack patterns is fun - you get to think about the abstract ideas that prom and drive the discovery of new exploits. Another useful trick is to add and remove meta-characters fro the beginning or end of 'fields'. In the lab we discovered that simply adding and removing a slash from filename induces quite different behavior. In some cases, the request is not processed. In some cases using a slash causes a much higher load than not using one.



Breaking a request into fields and adding trailing meta-characters

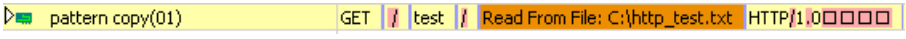
Conclusion: The end of field is a good place to try and confuse a parser.

Idea #9: Inserting invalid characters into fields

Many attacks take place over HTTP today and it will only get worse. Agents or processes at the termi end of GET/POST requests are handling user-supplied data. IDS systems and content filters must wa the fields and parameters being passed to CGI scripts to sniff out attacks. Consider that many attacks involve requests with large relative paths. Here in the lab we built a pattern to test just this problem. W store a whole set of possible CGI field names and pass them along with large relative paths:

```
GET /cgi-bin/test.cgi?MXT=student&SEM=MRC&DIV=0&LOG=any&
DAC=10&DEE=2000&FILE=/../../../../../../../../../../../
../WINNT/SYSTEM32/REPAIR/SAM._
```

It became apparent that it didn't matter how far out we escaped the path, once you hit the root of the filesystem you stay there. Simply by sending a large enough path, we are almost assured to reach the root of the drive.



Just reading the request field from a file.

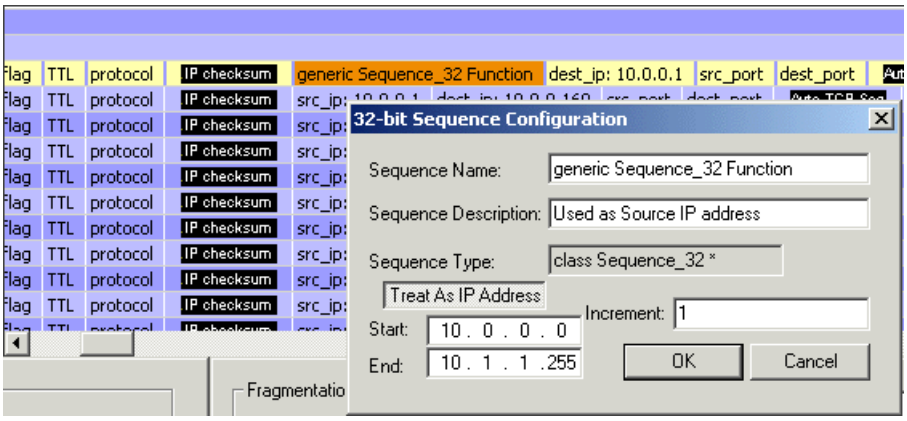
A test pattern can simply run a dictionary against the parameter list, inserting thousands of name/valu

pairs and watching the results. If the script understands what we send, it crafts an SQL statement and ships it off to the SQL server. At other times, the script fails if an invalid parameter is found. The IDS doesn't know this and throws a false positive. It is also possible the IDS will fail to recognize the attack when certain fields or characters are present. By inserting sets of meta-characters into each field we can confuse the IDS, crash the script, or cause a nonstandard SQL statement to be built.

Conclusion: Try every permutation. Be consistent and try everything.

Idea #10: Sensitive Dependance on certain Fields

Generally, you will find that certain fields are very important to the application request. We call these 'sensitive' fields. These fields are usually responsible for malloc's or offset calculation. Most denial of service attacks leverage sensitive fields to cause more resource usage. For example, a large buffer is sent that is very close to the upper limit. Think 'Ping of Death'. As we approach a border in the system (the buffer upper limit), the chances of a failure increase. We can cause 100% CPU usage by driving intrinsically large buffer allocation - or an infinite loop (or equivalent). At the edges of buffers we can find out off-by-one errors. At the edges of numerical ranges we can discover roll-overs. Y2K was a perfect example of this, and is the most widely known bug in the world.



Generating the request from a range of source IP's. Shown configuring a 32 bit sequence in the source IP field

Furthermore, by simply generating all of these requests from many source IP's, you can simulate DDOS attacks on your own servers and IDS systems. The load can be substantial when a sensitive field is leveraged.

Conclusion: Sensitive fields like hang out near the 'borders' (i.e., borders of ranges, borders of fields, borders of buffers... etc).

Closing

The recent Openhack.com incident brings up an interesting issue with current IDS systems. Most commonly, the IDS system is watching for known exploits to be executed. These are often the same known exploits that are tested for by commercial security scanners. For example, ISS sells both a security scanner and an IDS system. It is not unreasonable to assume that the vulnerabilities detected by their security scanner correspond to the attacks that their IDS system detects.

If a site administrator runs a security scanner on the network, and resolves all issues detected by it, the vast majority of attacks that an IDS detects will be failed attack attempts, while the ones that succeed likely be detected by neither the scanner nor the IDS. Therefore, while the IDS may detect 98 or 99% attack attempts, it is the 1 or 2% of attacks that are not detected that are particularly troubling.

The Openhack.com test site was compromised by exploiting a previously unknown exploit in the shopping cart software, something that is very unlikely to be detected by an IDS system (even if it wasn't tunneled through HTTPS), because none of their signatures match the attack. The PCWeek hacking challenge

the fall of 1999 fell in almost exactly the same manner. The hacker, JFS, exploited a previously unkno hole in the classified ads CGI script. This type of attack is most disturbing, because, in the case of the Openhack.com server, it was locked down by a professional security consultant. It is even less likely t a known exploit will succeed. The attacks that do succeed will usually be crafted by an individual who examined the software on the system and found a new hole.

The Lab

The methods and software used in this article has been developed into a package called 'Hailstorm'. software runs under Windows NT/2000 and generates spoofed TCP/IP transactions using a custom kernel driver. If your interested in beta-testing or evaluating Hailstorm, it can be downloaded from our web site at http://www.clicktosecure.com. The package is available for free and will include source co

Greg Hoglund is a software engineer and researcher. Hoglund's primary focus is kernel-based intrusion detection/prevention. Starting sever years ago in the "security scanner" market, Hoglund has moved on through a variety of research positions and is now a founder of Click To Secure, Inc., a lab-based company devoted to software reliability testing. Other projects include rootkit.com, a windows NT kernel-mode ro project, and speaking/training at various security conferences. He wrote a chapter on buffer-overflows recently published in 'Hack Proofing Network' (published by Syngress).