Secure Network Server Programming on Unix

Andreas Krennmair `ak@synflood.at`
Borland Software Corporation, Linz, Austria

December 2006

Contents

1 Introduction **1**

2 Design and Implementation **2**
2.1 General Coding Style 3

3 Attack Vectors **3**
3.1 Implementation Errors in trapdoor2 3
3.2 Implementation Errors in SSL/TLS 7
3.3 Denial of Service Attacks 9
3.3.1 Denial of Service Attacks on System Logging 9
3.3.2 Denial of Service Attacks on Subprocess Creation 10
3.3.3 Analyzing Statistical Distribution 11

4 Conclusions **12**

Abstract

This paper describes a software system to securely execute predefined commands over an untrusted network, using an authentication method and a measure of transport layer security. This software system – called "trapdoor2" – is used as an example to describe a number of programming techniques as countermeasures against potential attacks which can be considered "state of the art" in the field of programming secure network servers using the C programming language on Unix and Unix-like systems.

1 Introduction

When Clifford Wolf, the later co-author of trapdoor2[1], was once defining a security policy for a computer network connected to the internet, the demand for a way to temporarily "poke holes" into a firewall came up. The first solution was a simple, insecure telnet-based server running on Linux called "trapdoor" that one only had to connect to and enter a secret command to run a predefined command, i.e. to allow a connection from the client's IP address and any port to a certain server and a certain port for a certain time, e.g. 30 seconds. This allowed company employees who were

[1] `http://oss.linbit.com/trapdoor2/`

given the necessary commands to log into the company network even when they were working off-site, e.g. at a customer's site or at home.

Of course, this solution had a number of disadvantages, as an unencrypted ad-hoc protocol (telnet) was used to transmit sensitive information (the secret command). So an alternative solution had to be found for this issue, and an alternative system was created to address the issues mentioned. This new system – called "trapdoor2", and also implemented for Unix-like operating systems – was based on a robust and widely deployed standard protocol (HTTP) using a secure transport layer (SSL/TLS). Using such standard protocols made it easy to gain a better acceptance for trapdoor2, as SSL/TLS is generally considered secure and HTTP is a protocol that can be easily firewalled using e.g. proxies ("application-level firewalls").

Of course, the whole problem wouldn't be that easy if there wasn't a catch: a number of attack vectors on trapdoor2 do exist, and I will show in this paper how countermeasures for a number of them were implemented. I make no security claims on trapdoor2, but I do think that I can give an overview over the state-of-the-art methods in practical network server security on Unix-like systems.

2 Design and Implementation

As mentioned in the introduction, trapdoor2 was implemented on Linux, but basically runs on most other Unix-like systems. It was implemented as a stand-alone server to have total control over all aspects of the server, from accepting the network connections to handling authentication requests from the client to proper termination. I would like to explain both design and implementation by describing the typical program flow:

When trapdoor2 starts up, the first thing it does is that it retrieves its configuration by reading a configuration file. This file contains basic configuration values and commands and their associated secret command ("magic cookie" in trapdoor2 nomenclature). If configured, trapdoor2 starts as a daemon, i.e. puts itself into the background with no connection to any controlling terminal anymore.

When this is completed, it allocates a listening socket to be able to accept network connections from other hosts. Currently, trapdoor2 supports IPv4 and IPv6 sockets. Then it waits for incoming connections, which are then accepted and passed over to the request-handling part of trapdoor2.

Handling a request in trapdoor2 starts by creating a new subprocess while the "parent" process keeps on accepting new connections and creating new subprocesses. The subprocess then sets up the additional SSL/TLS security layer. When this is finished, the actual request is being received and parsed by the subprocess which then tries to authenticate the parsed magic cookie against the list of configured magic cookies.

In case a valid magic cookie was found, the associated command is executed and the exit status ("success" or "failure") is returned to the client that previously authenticated itself. If no valid magic cookie was found, a failure is indicated to the client. Then, the SSL/TLS connection is properly being terminated, the network connection gets closed, and the subprocess exits.

The program flow described above is straight forward, more or less obvious to developers with experience in the Unix networking field, and easy to implement. To quote H.L. Mencken, "For every complex problem, there is a solution that is simple, neat, and wrong."[2] Well, while trapdoor2 is definitely simple and neat, it is not nec-

[2] http://en.wikiquote.org/wiki/H._L._Mencken

essarily wrong, but its simple, straight forward, and probably naive design contains a number of obvious and not-so-obvious flaws that could be used to attack trapdoor2 and use it to break into the system where it is running. These flaws need to be addressed.

2.1 General Coding Style

During the development of trapdoor2, the source code has already been regularly checked for potentially insecure library functions, and after the major development effort was finished, the source was carefully audited.

The only calls of `strcpy` and `strcat` in the complete source can be found in the MD5 implementation which has been taken from the Apache library, and while it's not absolutely obvious from the code, it can be shown that the destination buffer of `strcpy` and the following operations on this buffer will not overflow.

Also, potentially insecure functions such as `sprintf` have been avoided, and `snprintf` has been used instead. All `*printf` variations have been checked for potential format string attacks.

In my opinion, which standard library functions should or should not be used is (or at least should be) more or less common knowledge, and so I won't go into this topic further, as it has been handled before in many publications, such as [Whe03].

3 Attack Vectors

An attack vector is a path or means by which an attacker can (potentially) gain access to a computer or a network. In the design described above, a number of attack vectors exist, which I would like to explain (including countermeasures).

3.1 Implementation Errors in trapdoor2

The first and most obvious attack vector is an attack on the HTTP implementation of trapdoor2 itself. Even while the HTTP stack was security-audited, there still exists the probability that some kind of security hole couldn't be found yet and can be exploited by an attacker. Such a security hole could e.g. be a buffer overflow that leads to arbitrary code execution.

As trapdoor2 runs with "root" privileges, this is a big threat. So, the first countermeasure is to drop the privileges of the subprocess. But this leads to the problem that the executed command wouldn't run with root privileges anymore, and thus wouldn't be able to e.g. modify the packet filter rules as it should. So, the solution is to create a first subprocess, which keeps root privileges, and then creates another subprocess which drops its privileges. These two subprocess communicate via a simple bidirectional communication channel (two unnamed pipes, to be precise) using a strictly defined and enforced communication protocol:

```
1 void handle_http_request(int fd, struct in_addr in)
2 {
3   int childpid, rc;
4   int c2p_fds[2];
5   int p2c_fds[2];
6   int pipe_fds[2];
7
8   client_ip = inet_ntoa(in);
```

```
9
10    rc = pipe(c2p_fds);
11    if (rc != 0) {
12      limit_syslog(LOG_ERR, "pipe failed: %s", strerror(errno));
13      return;
14    }
15
16    rc = pipe(p2c_fds);
17    if (rc != 0) {
18      limit_syslog(LOG_ERR, "pipe failed: %s", strerror(errno));
19      return;
20    }
21
22    childpid = fork();
23
24    if (childpid == 0) {
25      close(c2p_fds[0]);      /* child to parent: write only */
26      close(p2c_fds[1]);      /* parent to child: read only */
27      pipe_fds[0] = p2c_fds[0];
28      pipe_fds[1] = c2p_fds[1];
29      do_handle_http_request(fd, pipe_fds);
30      exit(EXIT_SUCCESS);
31    } else if (childpid == -1) {
32      limit_syslog(LOG_ERR, "fork failed: %s", strerror(errno));
33      return;
34    }
35
36    close(fd);
37    close(c2p_fds[1]);
38    close(p2c_fds[0]);
39    pipe_fds[0] = c2p_fds[0];
40    pipe_fds[1] = p2c_fds[1];
41
42    handle_client_request(pipe_fds);
43  }
```

The unprivileged subprocess handles the requests, parses the magic cookie, and hands it over to the privileged subprocess. Anything that looks suspicious to the privileged subprocess, like a very long magic cookie (longer than 100 characters) or a discrepance between the announced length and the actual length of the magic cookie received from the unprivileged subprocess leads to immediate termination, and the request won't be fulfilled: the privileged subprocess cannot trust the unprivileged subprocess (it could be under control of an attacker), everything received from the unprivileged subprocess is a potential attack and must be handled with extreme caution.

```
1   static void handle_client_request(int pipe_fds[])
2   {
3     int size, rc, ret = 1;
4     char buf[100], *cmd, *resp;
5
6     rc = (int)read(pipe_fds[0], &size, sizeof(size));
7     if (rc < (int)sizeof(size) || size>=100) return;
8     rc = (int)read(pipe_fds[0], buf, (size_t)size);
9     if (rc < size) return;
10    buf[size] = 0;
```

```
11
12      /* try to find command for the cookie */
13      if ( (cmd = get_command(buf)) ) {
14        /* Don't log this thru the limiter ! */
15        syslog(LOG_INFO | LOG_AUTHPRIV, "Running '%s' for %s.", cmd, client_ip);
16        run_command(cmd, client_ip);
17        ret = 0;
18      }
19
20      (void) write(pipe_fds[1], &ret, sizeof(ret));
21
22      if ( ret == 0 ) {
23        if ( (resp = get_response(buf)) ) {
24          ret = strlen(resp);
25          (void) write(pipe_fds[1], &ret, sizeof(ret));
26          (void) write(pipe_fds[1], resp, strlen(resp));
27        } else
28          (void) write(pipe_fds[1], &ret, sizeof(ret));
29        }
30    }
```

But these measures aren't enough to protect the system from an attacker: as long as the unprivileged subprocess still has complete access to the system's filesystem and an attacker can execute arbitrary code, the attacker can attack other system components. On Unix-like systems, so-called "SUID" binaries exist, which aren't run with the current user's privileges but with privileges of the user that owns that binary[Smi97]. Normally, such binaries are related to authentication and system administration, and the past has shown that such binaries are sometimes exploitable to gain root privileges on a system simply due to a programming error.

The countermeasure to this problem: restrict access to any other files. Fortunately, Unix-like systems provide a mechanism for that, called "chroot" (change root). Using it, the file system root that is visible to the subprocess is relocated to another directory which contains no files or any other ways to interact with the system, and the unprivileged subprocess has no ways of directly attacking any potentially vulnerable local program.

Besides that, there are other measures like stack execution protection and stack overflow protection. While these are outside the scope of trapdoor2, such measures are already implemented in operating systems like Linux and OpenBSD or as the *ProPolice* or *StackGuard*[WC03] extensions to the widespread *GCC* compiler suite, and provide an additional layer of security.

Dropping privileges Trapdoor2 does a number of things to protect the system from the unprivileged process. As a first step, it limits the CPU time that the subprocess is allowed to run. This effectively hinders the attacker to consume a significant amount of system resources and use the subprocess as e.g. network protocol proxy, client node for distributed computing project or zombie in a bot net.

```
1 struct rlimit rlim;
2
3 rlim.rlim_cur=rlim.rlim_max=2;
4 if (setrlimit(RLIMIT_CPU, &rlim)) {
5   limit_syslog(LOG_ERR, "setrlimit(RLIMIT_CPU) failed: %s\n", strerror(errno));
6   exit(EXIT_FAILURE);
```

```
}
```

Other limitations that are enforced are related to the maximum size of memory that can be allocated by the process.

```
rlim.rlim_cur=rlim.rlim_max=1024*512;
if (setrlimit(RLIMIT_DATA, &rlim)) {
  limit_syslog(LOG_ERR, "setrlimit(RLIMIT_DATA) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

rlim.rlim_cur=rlim.rlim_max=1024*64;
if (setrlimit(RLIMIT_STACK, &rlim)) {
  limit_syslog(LOG_ERR, "setrlimit(RLIMIT_STACK) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

rlim.rlim_cur=rlim.rlim_max=1024*512;
if (setrlimit(RLIMIT_RSS, &rlim)) {
  limit_syslog(LOG_ERR, "setrlimit(RLIMIT_RSS) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}
```

When this is done, the subprocess first enters the chroot environment, and then drops group and user privileges to an unprivileged user and group. As a last step, a timer is started which kills the process when the request couldn't be completed within 10 seconds of clock time.

```
/* create chroot_dir if it's not there, e.g. for /var/run subdirs
 * which are automatically removed at system reboot. */
(void) mkdir(chroot_dir, 0755);

if (chdir(chroot_dir)) {
  limit_syslog(LOG_ERR, "chdir(chroot_dir) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

if (chroot(chroot_dir)) {
  limit_syslog(LOG_ERR, "chroot(chroot_dir) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

if (setgid(chroot_gid)) {
  limit_syslog(LOG_ERR, "setgid(chroot_gid) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

if (setgid(chroot_uid)) {
  limit_syslog(LOG_ERR, "setuid(chroot_uid) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}

if (alarm(10)) {
  limit_syslog(LOG_ERR, "alarm(10) failed: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}
```

This whole concept of having a totally unprivileged subprocess with a very tight and well-defined interface to the privileged process is known as “privilege separation” and has been first implemented in OpenSSH[PFH03].

3.2 Implementation Errors in SSL/TLS

Recent history in network computer security has shown that, while providing cryptographic security, existing SSL and TLS implementations were vulnerable to a number of different attacks on the SSL/TLS protocol level. A quick search on `http://secunia.com/` leads to a list of 8 security vulnerabilities of OpenSSL[3] in the last few years, and a number of vulnerabilities for commercial SSL/TLS implementations. This clearly shows that SSL/TLS is an attack vector[Mur03].

With the measures we've described above, this actually shouldn't be much of a problem, but its needs to ensured that no interaction between the attacker and SSL/TLS can happen before the second subprocess dropped all its privileges. For this, the SSL/TLS initialization routine has been divided into the two phases: the first phase is being executed when the second subprocess is still running with root privileges, which reads in cryptographic keys and certificate files. Then, the subprocess drops all its privileges, and when this is completed, the second phase initialization of SSL/TLS (i.e. the SSL/TLS handshake, where the first interaction between a potential attacker and the SSL/TLS implementation can happen) is done. This makes sure that there is no chance to exploit any SSL/TLS security hole in the subprocess when it still has its root privileges.

```
1  static void do_handle_http_request(int fd, int pipe_fds[])
2  {
3    /* ... */
4    init_ssl();
5    drop_privileges();
6    init_ssl2(fd);
7    /* ... */
8  }
9
10 /* ... */
11 void init_ssl(void)
12 {
13 #if HAVE_LIBSSL
14   SSL_load_error_strings();
15   SSLeay_add_ssl_algorithms();
16   meth = SSLv23_server_method();
17   ctx = SSL_CTX_new(meth);
18   if (ctx == NULL) {
19     limit_syslog(LOG_ERR, "SSL: failed to create new context");
20     exit(EXIT_FAILURE);
21   }
22
23   if (SSL_CTX_use_PrivateKey_file(ctx, ssl_keyfile, SSL_FILETYPE_PEM) <= 0) {
24     limit_syslog(LOG_ERR, "SSL: failed to use key file %s", ssl_keyfile);
25     exit(EXIT_FAILURE);
26   }
27
```

[3] A popular SSL/TLS implementation on Unix-like systems

```
28    if (SSL_CTX_use_certificate_file(ctx, ssl_certfile, SSL_FILETYPE_PEM) <= 0) {
29      limit_syslog(LOG_ERR, "SSL: failed to use certificate file %s", ssl_certfile);
30      exit(EXIT_FAILURE);
31    }
32
33    ssl_handle = SSL_new(ctx);
34    if (ssl_handle == NULL) {
35      limit_syslog(LOG_ERR, "SSL: failed to get new handle");
36      exit(EXIT_FAILURE);
37    }
38
39  #endif
40
41  #if HAVE_LIBGNUTLS
42    gnutls_global_init();
43
44    gnutls_certificate_allocate_credentials(&x509_cred);
45
46    gnutls_certificate_set_x509_key_file(x509_cred, ssl_certfile,
47                                         ssl_keyfile, GNUTLS_X509_FMT_PEM);
48
49    generate_dh_params();
50
51    gnutls_certificate_set_dh_params( x509_cred, dh_params);
52
53    session = initialize_tls_session();
54  #endif
55  }
56
57  void init_ssl2(int fd)
58  {
59    int rc;
60  #if HAVE_LIBSSL
61    SSL_set_fd(ssl_handle, fd);
62    rc = SSL_accept(ssl_handle);
63    if (rc == -1) {
64      limit_syslog(LOG_ERR, "SSL: failed to accept connection");
65      exit(EXIT_FAILURE);
66    }
67  #endif
68  #if HAVE_LIBGNUTLS
69    gnutls_transport_set_ptr( session, (gnutls_transport_ptr)fd);
70    rc = gnutls_handshake(session);
71
72    if (rc < 0) {
73      exit(EXIT_FAILURE);
74    }
75  #endif
76  }
```

As you can see in the above code examples, trapdoor2 supports both OpenSSL and GNU TLS. This gives the system administrator the choice between two different SSL/TLS implementations. This is an admission to the fact that monocultures are generally bad in computer networks when it comes to security.

3.3 Denial of Service Attacks

Another attack vector on trapdoor2 are Denial of Service (DoS) attacks. Although this kind of attack won't let an attacker execute arbitrary code on the system, it can still render the service useless, so that legitimate users can't use trapdoor2 anymore in the intended way (at least for the time of the attack). Trapdoor2 contains countermeasures against this kind of attacks on two potentially vulnerable points.

3.3.1 Denial of Service Attacks on System Logging

The possibility exists that an attacker might be able to take over the unprivileged subprocess. In this case, there would be still one way to "inject" data into the system, and that is the system logging (syslog) facility provided by the operating system. So, an intruder might still fill up the hard disk with data, and temporarily interrupt normal system operation. In order to slow down such an attack, the maximum number of system logging messages is limited.

```
1  #define MSG_PER_SECS 0.1
2  #define MAX_BURST_LEVEL 30
3  #define PENALTY_LIMIT 10
4
5  static float msg_counter = MAX_BURST_LEVEL;
6  static time_t last_message = 0;
7  static int penalty_mode = 0;
8
9  void limit_syslog (int pri, const char *fmt, ...)
10 {
11   va_list ap;
12   time_t now = time(0);
13
14   if ( last_message ) {
15     msg_counter += (now-last_message) * MSG_PER_SECS;
16     if ( msg_counter > MAX_BURST_LEVEL ) msg_counter = MAX_BURST_LEVEL;
17   }
18   last_message = now;
19
20   if ( msg_counter <= 0 ) {
21     if ( ! penalty_mode ) {
22       syslog(LOG_WARNING | LOG_AUTHPRIV, "Too many syslog messages (DOS Attack ?)");
23       syslog(LOG_WARNING | LOG_AUTHPRIV, "... going to be silent for %d seconds.",
24                  (int)(PENALTY_LIMIT/MSG_PER_SECS));
25     }
26     penalty_mode = 1;
27   }
28
29   if ( msg_counter >= PENALTY_LIMIT ) penalty_mode = 0;
30   if ( penalty_mode ) return;
31
32   va_start(ap, fmt);
33   vsyslog(pri | LOG_AUTHPRIV, fmt, ap);
34   va_end(ap);
35
36   msg_counter--;
37 }
```

Currently, this limitation is only in effect when the logging function that enforces the limitation is called. This means that direct calls to the syslog function are not limited, which – although it requires the injection of custom malicious code into trapdoor2 – is a potential way of circumventing this protection.

3.3.2 Denial of Service Attacks on Subprocess Creation

The easiest way of starting a denial of service attack against the machine where trapdoor2 is running on is to open a lot of connections simultaneously. When trapdoor2 accepts each of these connections, two child processes will be started for each open connection, quickly increasing the system load on the machine. Thus, trapdoor2 implements a connection limiting mechanism to limit the number of incoming connections per source IP, making it virtually impossible to increase the system load while still being functional for users from other source IPs.

This mechanism works as follows: trapdoor2 computes a hash from the source IP, also taking into account a random value that changes every 8 seconds (the time window) to make the resulting hash virtually unpredictable for the attacker. This means that the hash for the same IP address changes every 8 seconds.

The result of *hash mod SLOTNUMBERS* is then used to determine the connection counter slot number. In the determined slot, the counter is incremented, and if it peaks at the maximum number of connections per slot and time window, the connection is closed immediately, otherwise the connection is handled as usual. As soon as the next time window begins, the counter for each connection counter slot is reset to 0.

With a current slot number of 397 and a maximum of 12 connections per 8 seconds, this means that at most 4764 requests per 8 seconds will be handled, but only if the attacker manages to run a distributed denial of service attack where the hashed source IPs lead to slot numbers completely ranging from 0 to 396.

```
1  char *addr_data = (void *) &client.sin_addr;
2  time_t current = time(NULL) & ~7;
3  unsigned int slot;
4
5  if ( current != last ) {
6    if ( randomfd < 0 || read(randomfd, &modval,
7      sizeof(modval)) != (ssize_t)sizeof(modval) ) modval = random();
8    last = current;
9  }
10
11 slot = dohash(addr_data, (unsigned int) sizeof(struct in_addr),
12                 (unsigned int) modval) % MAX_CONN_PRIME_SLOT_NUMBER;
13
14 conn_counter[slot].counter =
15     conn_counter[slot].last == current ? conn_counter[slot].counter + 1 : 0;
16 conn_counter[slot].last = current;
17
18 if (conn_counter[slot].counter >= MAX_CONN_PER_SLOT_AND_8SECS) {
19   close(connfd);
20   continue;
21 }
22
23 limit_syslog(LOG_INFO, "Connection from %s:%u [mod=%08X, slot=%u, count=%u]",
24                 inet_ntoa(client.sin_addr), ntohs(client.sin_port),
25                 modval, slot, conn_counter[slot].counter);
```

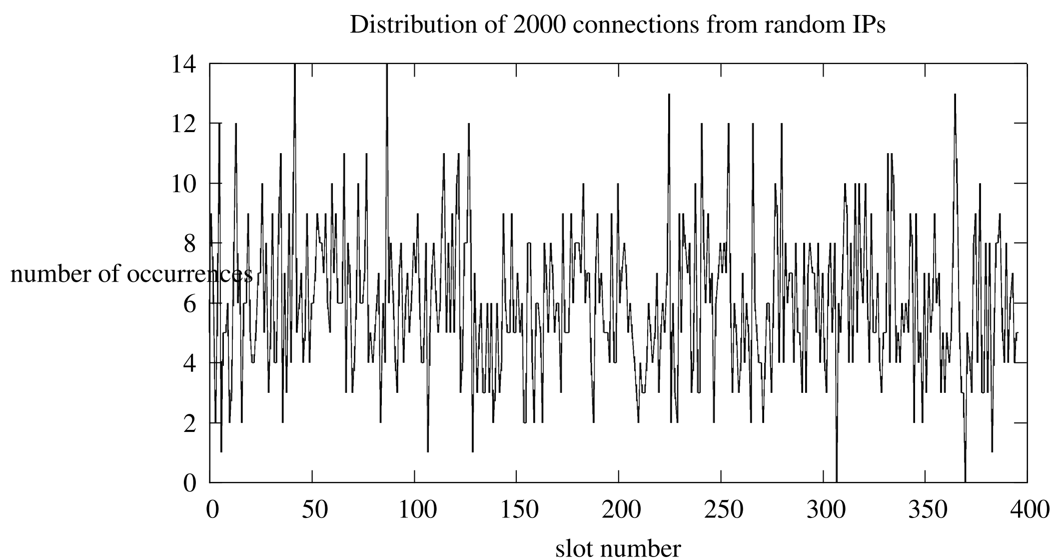

Figure 1: Distribution of 2000 connections from random IPs

The hashcode itself is defined as the following function $h(n)$ on a character array A with n elements. r represents the random value that changes every 8 seconds:

$$h(0) = r \tag{1}$$

$$h(n) = (2^{h'(n-1)}h(n-1) \bmod 2^{32}) \oplus (\frac{h(n-1)}{2^{32-h'(n-1)}} \bmod 2^{32}) \oplus A_n \tag{2}$$

$$h'(n) = h(n) \bmod 7 + 9 \tag{3}$$

In fact, any more or less secure hash algorithms like MD5 or SHA1 could have been chosen, but the hash algorithm above is a lot more compact and easier than those "standard" algorithms and still delivers a good distribution for changing values of r.

3.3.3 Analyzing Statistical Distribution

To show how the crucial part (i.e. the hashing algorithm) of the connection limiting code works, I generated two graphs. The first one shows the distribution of 2000 connections from randomly-generated IPs (Figure 1). While the result is not perfect, it looks well-distributed, and in my opinion, "good enough".

The second one shows the distribution of connections from a /20 subnet (Figure 2). While not as "good" as the first one, it still shows an acceptable behaviour for a case where an attacker has gained control over a whole subnet to use it for DoS attacks.

And it also exposes an interesting paradoxon: even while the algorithm does not provide equal distribution, this makes it possible that actually *less parallel connections* are accepted from the incriminated subnet. If you make the assumption that there are no legitimate users in a compromised subnet, this observed property of the algorithm reduces the effectiveness of a DoS attack from a subnet.

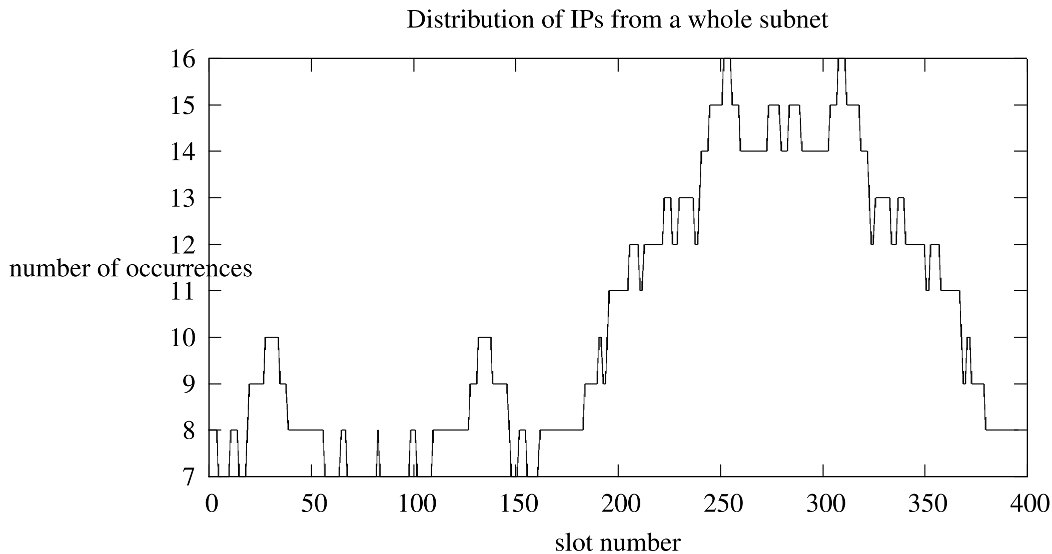

Figure 2: Distribution of connections from a /20 subnet

4 Conclusions

In this paper, I gave some insight on the implementation of trapdoor2, a secure network server for executing remote commands using a secure authentication mechanism. I showed a number of attack vectors against trapdoor2, including solutions to address potential attacks via these vectors.

A few wise words: audit your applications regularly. Let several people audit them. Don't solely rely on programmatic mechanisms within your program, care about "defense in depth". And don't forget: nobody's perfect. Shit does happen. Murphy is omnipresent.

References

[Mur03] MURPHY, Keven: Traveling Through the OpenSSL Door. (2003). http://www.giac.org/practical/GCIH/Keven_Murphy_GCIH.pdf

[PFH03] PROVOS, Niels ; FRIEDL, Markus ; HONEYMAN, Peter: Preventing Privilege Escalation. In: *12th USENIX Security Symposium* (2003). http://www.citi.umich.edu/u/provos/papers/privsep.pdf

[Smi97] SMITH, Nathan P.: Stack Smashing vulnerabilities in the UNIX Operating System. (1997). http://www.weycrest.co.uk/information/infosec/info/security/buffer-alt.pdf

[WC03] WAGLE, Perry ; COWAN, Crispin: StackGuard: Simple Stack Smash Protection for GCC. (2003). http://gcc.fyxm.net/summit/2003/Stackguard.pdf

[Whe03] WHEELER, David A.: Secure Programming for Linux and Unix HOWTO. (1999-2003). `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html`