Threads Against Delay

Improved programming techniques for dictionary attacks over the net

Sebastian Krahmer krahmer@cs.uni-potsdam.de

Abstract

Recently there was a posting on a major security mailinglist claiming that a networkservice was vulnerable to an attack because it's implementation didn't used `sleep()` after a failed password-authentication. The security people found that it was therefore possible for an attacker to use the dictonary-attack over the net and fixed the bug.

1 Introduction

Allmost all user-bound Internet services require some kind of password-authentication. Usually this is a user/password pair as used by POP3, FTP, TELNET, IMAP ... protocols.
When a wrong user/password pair is given in the authentication phase the service usually delays execution to make password guessing harder. Additionally, they shut down the connection after three unsuccessfull attempts to make it even harder for the attacker to guess passwords.
These services work usually concurrently, i.e. they allow multiple sessions at the same time. This has the benefit of allowing more than one user to read his mail for example. I will show that this benefit has its drawbacks by discussing old and new ways of attacks and by giving you a working program at the end of this discussion which shows that nowadays this way of authentication is outdated.

2 How the services work

There are a lot of ways these parallelism could be implemented. I'll name some of them, the most common ones. The list is in no way complete but shows you how most of the UNIX systems handle this. For our reason it does not really matter how the target handles its multisession-capability - as long as there is one.
It might be a problem that some approachs are more vulnerable to Denial Of Service (DoS) attacks than others.
For a more in-depth description and analyzation of these and even more techniques, please refer to [Stevens] and [Christiansen].

2.1 Inetd approach

Using the inetd approach, the sysop sets up the program `inetd` with the configfile `/etc/inetd.conf` where all services are named which are offered by the system. This could include FTP, POP3 and TELNET for example. The `inetd` superserver binds to the specified ports and ensures that clients get the right service when connection takes place. For that matter the superserver creates a new process and executes the specified program. This program is connected via `stdin`, `stdout` and `stderr` to the network and is ready to serve the protocol now. The old process just loops and listens for incoming connections again.
For low-traffic and/or systems with a small user-base this is a good solution. However when connections come in fast you will have the overhead of executing the specified program every time which makes the system very slow. To avoid this, you usually set up standalone servers which are dedicated for just one service each.

2.2 Standalone servers

In this case a special program (for example `ftpd` is executed which does all the network-handling alone as well as ensuring the multisession capabillities:

```
while ((afd = accept(fd, NULL, 0)) >= 0) {
if (fork() != 0) {
close(afd);

while (waitpid(-1, NULL, WNOHANG) > 0);
continue;
}
/* child code */
...
```

In this sample-code [1] you see that every arrived connection returned by `accept()` is passed to the child which is created by a call to `fork()`. The parent reapes all pending so called zombie-processes and continues waiting for new incoming connections. This way, when just connected to the server we can immediately open a new connection. We'll make heavy use of this feature later.

3 The authentication

After forking off the child, it is up to the child to look whether or not the requesting client is allowed to access

[1] Yes, it is really simple because I neither check for errors nor set signal handler for SIGCHLD signal. The code is ripped from my bserv brute-benchmark server.

the offered resources. So it usually begins to wait for a login/password pair. When it received the information it compares this with the entries from a given database. Here it is not important how this is done, neither is that the login information is transmitted in cleartext. Important is what the child does when the authentication failes. Usually it will invoke a `sleep()` call which puts the calling process to sleep for the given amount of seconds. In living code this looks somewhat like this:

```
fprintf(f, "100 Delay testserver ready.\r\n");

fgets(buf, sizeof(buf), f);
fprintf(f, "100 Hello. Expecting password\\
 now.\r\n");

fgets(buf, sizeof(buf), f);

if (strncmp(buf, secret, strlen(secret)) != 0) {
sleep(delay);
fprintf(f, "300 Wrong password.\r\n");
} else {
fprintf(f, "230 You won!\r\n");
printf("We have a winner!\n");
}
```

Here you see how the child handles the passed connection. At first it says something like *Hello I am here* to the client and expects the authentication to go on. In this very simple example it just waits for a password. The sent login-string is read in but is thrown away afterwards. After it has obtained the password it compares it with a hardcoded secret [2]. The following behavior now is important: When the right secret was sent, it just says *OK* and accepts the connection. If the password is not OK, then it sleeps a given amount of time which is usually three seconds. After the delay it complains that the given password is wrong. Note, that we found an important and interesting behaivior here: The RTT [3] changes when we give it a wrong password to eat! Keep this in mind because we'll use this later for our attack. Beside that, the server optionally drops the connection after failure or after an allowed maximum of unsuccessfull attempts.

4 The attack

As shown, both (the most common ones) ways work concurrently and sleep some seconds after false tries. If they continue as normal i.e. there is no `sleep()` call, then this will not hit us much. Indeed we will see that it is almost the same whether or not the server uses `sleep()` and/or drops connection.

[2]This is just an example. In real world the program would need to compare the encrypted password with the content of the `/etc/passwd` (or `/etc/shadow`) file of the system. From our point of view this simple `strcmp()` does its job well.

[3]We define the Round Trip Time as the time it takes to send a string to the server and get the reply during the protocol.

4.1 The common way

The common way now is to write a program or script which constantly opens connections to the server, tries a login/password pair from a dictionary and collects the result. Upon failure, it connects again until it succeedes. It is easy to see that with the given architecture of delay and connection-drop we do not have a chance to guess the right password.

4.2 The new way

What we now do is thinking about a concurrent client. A client that (as the server does) spawns a new thread [4] for each try (that is connection). With this technique we would save much time, because we can guess e.g. 20 passwords in parallel where we just had one guess before. Note, that this is only possible because the server works concurrently. This solution works and is easy to implement because the only resource you need to synchronize is the dictionary. On the other hand, you run e.g. 20 instances of your program when you guess a login this way.

5 Improving the new way

I have seen such programs, I even spent ideas and code to them. They work, but I was sure even they could be improved. So I started thinking again ...

5.1 Producer/Consumer

The first idea creeping in was to split up the task of connecting to the server from the task of guessing. A first implementation of the new dictionary attack run two threads, one of them constantly opening new connections to the server, the other one consuming these connections for password guessing. As you see, this is a classical producer/consumer problem: We have a shared resource which is a list or a set of open connection to the server. The concurrency is also classically solved, using mutexes and a signal/wait mechanism. To make sure there are always open connections for the consuming thread, we define a threshold constant of `MINCONN`, where the producing thread begins to open connections to the server when the counter is less than this constant. On the other hand, the consumer goes to sleep when it reaches the maximum count of `MAXCONN`. Through the hole project I used *POSIX threads* for the concurrency. I would refer to [Bradford] for starters in this topic.

As a result of this technique we have virtually one open connection which is never dropped. No matter whether or not the server drops connection, we just try this connection once and immediately pick a new one from the list. This won't cost us more time, as the producing thread opens new connections anyway.

We bypassed the *drop connection after failure* policy. A good start.

5.2 Managing the guesses

The problem with the above solution was that the consuming thread is blocked in the guessing-task which could take

[4]A thread is basicly the same as a process, but it's spawned much faster and it is easier to handle shared resources with. For that reason I use threads instead of processes in my program.

a few seconds due to the delay. This costs us a lot of performance. To solve this dilemma, we just create a new thread which obtains the task of talking the protocol with the server. This thread will also notify us when we hit the right login/password pair and is responsible for freeing any ressources which are used later. By now we have two threads, one constantly opening new connections, one taking these connections, and additionally for each try a new thread which dies after failure. We are close to breaking the delay.

5.3 The final cut

Until now we are good but we are just a few points better than our first solution which just connected in parallel and just synchronizes dictionary access. To increase speed again, we remember that after a sent login the answer comes immediately but after a wrong password the answer is delayed. This is the point where the idea of delaying breaks: It makes no difference for us whether or not the server invokes `sleep()` because we do not have to wait until it sais "wrong". We already know it is wrong when the answer does not come shorthand. This is possible because we already have the average time an answer takes by recording the time the send-login-get-answer procedure takes! We just time out when this time is overridden and take it for a failure. Since this time is calculated new for each connect we even include the current speed of the network. By adding a grace of, lets say, one second we make sure that we don't accidently skip a successfull try. In code it looks like this:

```
struct timeval tv1, tv2;

if (gettimeofday(&tv1, NULL) != 0)
die("talk::gettimeofday");

fprintf(f, "USER %s\r\n", user); fflush(f);
fgets(buf, sizeof(buf), f);

if (gettimeofday(&tv2, NULL) != 0)
die("talk::gettimeofday");

// calculate latency of server. ugh.
tv2.tv_sec  = tv2.tv_sec-tv1.tv_sec;
tv2.tv_usec = (tv2.tv_sec*1000000+
(tv2.tv_usec-tv1.tv_usec))%1000000;

// add grace
tv2.tv_sec++;
...
```

It is easy to see that we get the current time before and after the we send the login. The difference in time (plus grace) is the maximal amount of time we accept. If no answer arrives after we sent the password, we failed.

6 Breakfast

Using this technique we reach a very high rate. Using my reference-implementation available on [Krahmer] with 25 open connections and a threshold of five, I reached a rate of four guesses per second via a 33.6 PPP link. Against a 2 CPU SMP testserver with 256 MB of RAM and a low-price attacking machine I even reached 24 guesses per second with 40 parallel connections and a `MINCONN` of 25. That sounds impossible with a delay of five seconds first. Even a delay of 10 seconds will not make the results differ. Depending on how fast the server can handle incoming connections, the amount of free process-slots and the bandwith, there is nothing that forbids us to use 100 parallel connections.

7 Defense

As seen during the last chapters, the `sleep()` call is no defense against brute forcing. One way to prevent such attacks are strong passwords, which is perfectly impossible with a large userbase. Someone (most people say 25%) will have a weak password, and the attackers will find out who. You can try to only allow a maximum of, lets say 5, sessions from the same source which will open your server for a DoS attack if your legitimate clients are originated from a masquerading router as well as the attacker. In this case your security will also depend on the fact that the attacker does not have enough ressources, which is a weak base. Attackers will solve this restriction by defining IP-aliases for their network devices, thus coming from a different IP each try. However, it is probably a better solution than to just calling `sleep()` without source-restriction.
The described technique works against weak passwords. Weak passwords are usually choosen by the user itself, as long as you do not force strong ones. Alternatives (if not even the future) for authentication are RSA authentication, challenge-responce algorithms and client certificates as described in [Rescorla] and [Schneier]. Such algorithms can be used without much knowledge of the application programmer, he can use available frameworks such as PAM or OpenSSL which do all the authentication for him. PAM and OpenSSL are available on most UNIX systems meanwhile. Please note that even if you are using OpenSSL with client certificates these certificates must be protected using a passphrase. You should carefully choose the phrase as well as you should do with your normal passwords. The described authentication mechanisms are none better than plaintext passwords if you do not force strong phrases. This is even worse because stealing certificates does not require root access as stealing `/etc/shadow` does.
PAM may even be used to enforce strong phrases or passwords. If you use login/password authentication, use these possibilities and run password-crackers over your password-base to find weak passwords yourself, before crackers do.

8 Releated

A number of excellent papers and programs already exists on the topics of password-cracking and brute-forcing. The *john* password cracker [Solar] for example may be used to find weak passwords in given files. *John* can be used to create a stream of carefull created passwords printed to `stdout`. This stream can be taken by `TAD` which uses the given input instead of a dictionary. They work hand in hand. Another multithreaded bruteforcer is *bruth* [Fyodor], which uses the *just synchronize dictionary access* technique described above and works surprisingly fast.

References

[Christiansen, 1999] Tom Christiansen and Nathan Torkington *Perl Cookbook* Solutions and Examples for Perl Programmers O'Reilly 1999

[Bradford *et al.*, 1998] Bradford Nichols, Dick Buttlar and Jaqueline Proulx Farrell *Pthreads Programming* A POSIX Standard for Better Multiprocessing O'Reilly 1998

[Stevens, 1998] W. Richard Stevens *UNIX Network Programming* Volume 1, Networking APIs: Sockets and XTI Prentice Hall 1998

[Schneier, 1996] Bruce Schneier *Angewandte Kryptographie* Protokolle, Algorithmen und Sourcecode in C Addison Wesley 1996

[Rescorla, 2000] Eric Rescorla *SSL and TLS* Designing and Building Secure Systems Addison Wesley 2000

[Solar, 2000] Solar Designer *John the Ripper* A fast password-cracker using dictionary attack against a given file http://www.openwall.com/john

[Fyodor, 2000] Fyodor Yarochkin *bruth* A multithreaded bruteforcer, like tad. http://bruth.sourceforge.net

[Krahmer, 2000] Sebastian Krahmer *TAD* A multithreaded bruteforcer; described throughout this paper http://www.cs.uni-potsdam.de/homepages/students/linuxer