In remembrance of timing attacks

Stavros Lekkas, Thanos Theodorides

Difficulty

●●○

The purpose of this article is to bring back to the stage the case of the execution path timing analysis of UNIX daemons. This case has been initially addressed by Sebastian Krahmer, a SuSe employee, who has also published an article back in 2002 explaining the relative issue. We describe how to perform timing analysis over the execution path of a program in order to identify valid usernames on UNIX services.

Moreover we propose some methods for eliminating such issues from one's system. Time analysis of computational tasks is a topic on which extensive research has been made for various issues. In general, it is about calculating the time complexity of an algorithm in order to extract specific functionality-related results. Such results can be the timespan the program requires to produce output for specific input on a specific hardware platform or the Worst Case Execution Time (WCET). These kinds of details are crucial for real-time systems which demand accurate and fast response time as well as for embedded systems or even normal PC. However, time analysis is not performed only by software developers. Hardware vendors, for example CPU or graphics-chips manufacturers, focus on analysing the response time of their products since this is the primary feature that makes them competitive.

Timing analysis is getting more and more popular for its efficiency in the IT-security world too. Researchers have found ways to detect sophisticated kernel backdoors via timing analysis, based on the fact that programmes infected with malicious code execute more instructions -thus creating greater time complexity- than a normal version of the same program. Furthermore, analysis of how much time a program takes to produce output for different kinds of input (such as input of an existent user and input of a non-existent user) can reveal sensitive information of the system configuration to possible attackers providing more attack vectors.

The purpose of this article is to explain how a timing attack can be performed in order to reveal the sensitive information mentioned above and make guesses on whether a username in the system is valid. In addition to that, a prototype probing utility will be coded for the experiments constituting the practical part.

What you will learn...

- how to make valid assumptions by performing timing analysis over the execution path of a program,
- how to identify valid usernames.

What you should know...

- elementary C programming,
- elementary statistics.

Listing 1. *The implementation of `calc_time()'*

```
/* 0: */   long calc_time(char *username)
/* 1: */   {
/* 2: */   int n;
/* 3: */   struct timeval tvalue1, tvalue2;
/* 4: */   struct timezone tzone1, tzone2;
/* 5: */
/* 6: */   CLEAR(wBuf);
/* 7: */   gettimeofday(&tvalue1, &tzone1);
/* 8: */   snprintf(wBuf, sizeof(wBuf) - 1,
        "USER %s\r\n", username);
/* 9: */   write(socket_fd, wBuf, strlen(wBuf));
/* 10: */   CLEAR(rBuf);
/* 11: */   n = read(socket_fd, rBuf, sizeof(rBuf) - 1);
/* 12: */   gettimeofday(&tvalue2, &tzone2);
/* 13: */   return (tvalue2.tv_usec - tvalue1.tv_usec);
/* 14: */   }
```

What is an execution path timing analysis

A computer program is defined as an organized list of instructions that, when executed, cause the computer to behave in a predetermined manner. Being more specific, a program is a procedure that when receives the same input it will return the same output. The time required to produce this output is referred as time complexity of the program and this complexity is expected to be about the same for programmes that run in the same environment and they are given the same input.

As we all know, the flow of execution of each program is continuously changing in order to handle the given input correctly. Programming statements like `if-then-else` and `switch`, create imaginary -yet possible- crossroads that affect the order of instructions being executed. Each different way of terminating the program with a valid output creates an execution path. Of course, if the program consists of millions of lines of code, there can be billions of different execution paths. Each of these execution paths has its own set of instructions to perform, so it requires a specific amount of time to execute. The section of algorithm development science that has to do with calculating this amount of time is called Execution Path Timing Analysis (EPTA).

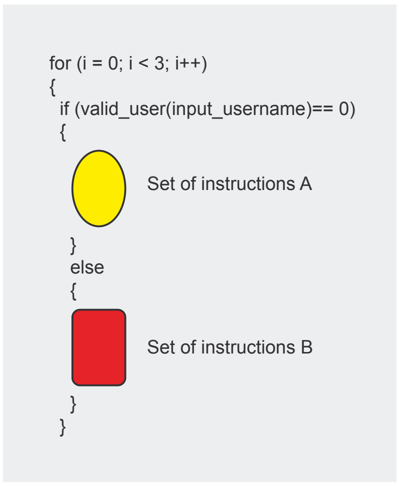
In order to understand EPTA, consider the code of an imaginary authentication mechanism as in figure 1.

The mechanism gives three chances to a user who wants to authenticate on the system, providing a password. Function `valid_user()` returns `0` if the user does not exist and `1` if the user exists. The two different geometrical shapes in figure 1 represent the different set of instructions executed when a user is valid or not. For example if the user does not exist, the set of instructions A is executed and then the for-loop proceeds one more time. In set A, actions like syslogging the unsuccessful login attempt or deactivating the users account may take place as the login failed. If the user exists, set of instructions B is executed and actions like binding on a shell or logging a successful login might occur. However if wrong password is encountered, the for-loop proceeds one more time etc. Figure 2 depicts the control-flow graph of the code of figure 1. A combination of all possible execution paths is presented in figure 3.

Consider the following events taking part in a case scenario using the authentication mechanism we mentioned above:

- auth-mechanism prompts for username and we enter an invalid username. Auth-mechanism prompts for password, we enter a random password (does not matter since user is invalid and authentication will fail anyway) and we get username prompt again (`i == 1` in for-loop),
- we enter a valid username so we get a prompt for password. We enter a wrong password for this username. Authentication fails and we get the username prompt again (`i == 2` in for-loop),
- we enter a valid username and a valid password. Authentication succeeds.

Combining figure 3 and table 1 and based on the above scenario we can perform time analysis of the execution path (see figure 4). A different scenario will demand a different execution path so time analysis will result into a different time value. The worst that can happen in a real case scenario is that a possible attacker may be able to reconstruct the precise execution path, collect response-time statistics and proceed to a timing attack.



```
for (i = 0; i < 3; i++)
{
 if (valid_user(input_username)== 0)
 {

  }
  else
  {

  }
}
```

Figure 1. *Imaginary authentication code*

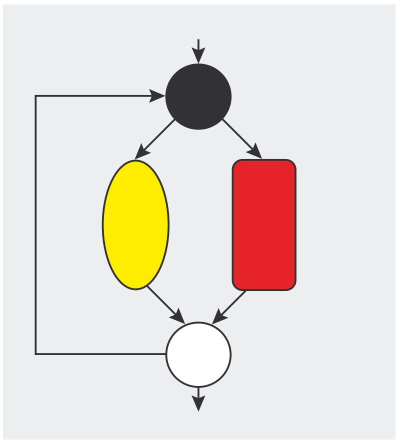


Figure 2. *Control-flow graph*

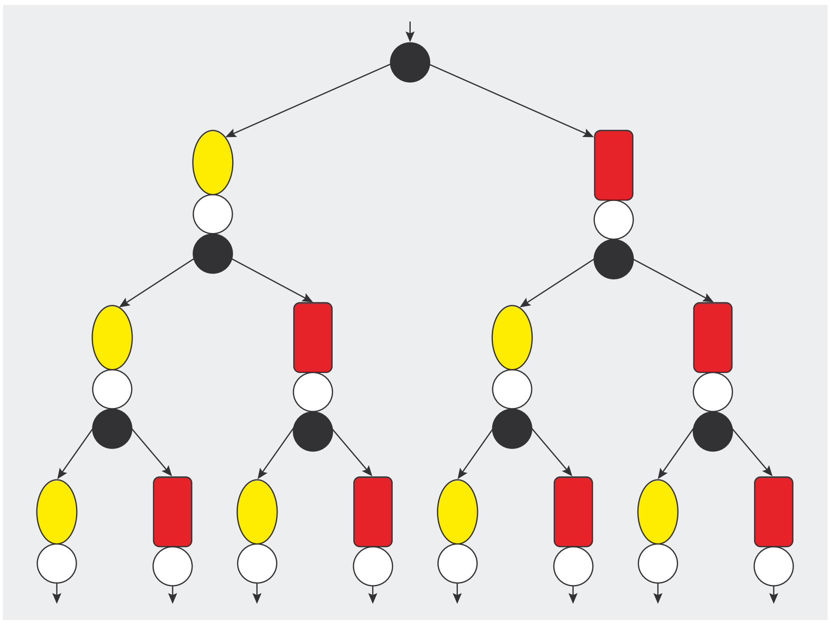


Figure 3. *Path explosions after three iterations (worst case scenario)*

What is a timing attack

A timing attack is a practical method under which the attacker attempts to extract information by analyzing the time taken (up to a desirable precision) to execute specific parts of an algorithm. The efficiency of this attack resides on the fact that every operation in a computer takes time to execute. The information leakage from a system can be made possible through measurement of the time the system takes to respond to certain queries. Note that if an algorithm is implemented in a way that every subroutine takes the same time to return results, a timing attack is impossible. In reality something like that is almost infeasible as most implementations *sacrifice* the security of the algorithm in order to have quicker response times in average (which is more desirable by software vendors). Services like ftp, telnet, OpenSSH and probably every service that uses Pluggable Authentication Modules (PAM), which are not implemented with the timing attack possibility in mind, are vulnerable. Keep in mind that although many of these services are secure up to a point (correct input validation, efficient memory management), the vulnerability, if any, of a timing attack, resides into the implementation.

A timing attack can be really useful when trying to discover the existence of a user on a remote system. According to Krahmer (see Evaluating Krahmer's work on EPTA), services like the ones mentioned above classify the login types as:

- *Valid Login*: If both the username and password are valid, the user authenticates and additional instructions get executed, e.g. a shell is executed, a directory structure is being printed or a 2nd authentication mechanism appears,
- *Valid Login with restrictions*: The user exists but although username and password are correct, the user is not allowed to login. This is possible if his account is suspended, expired or because he is listed in a deny file. For example he may be allowed to use ftp but not sshd,
- *Invalid Login*: The user does not exist. However, the service still requires a password from him so that a possible attacker does not know that the user is indeed invalid,
- *Special Login*: Superuser logged in and apart from a shell, some additional features were executed (additional overall time).

The above classes of logins are not handled the same way by every service. For example some ftp servers may execute more code for an invalid login rather than for a valid, while some ssh servers may do the opposite. So assumptions like *response time was quicker for this username so it must be valid*, are false. In order to make clear and correct assumptions a special sequence of tests is necessary. This sequence, most of the times, is sufficient:

- try to login with a valid username for quite a few times. For each login try, measure the time elapsed to be prompted for a password,
- repeat the above procedure but this time picking up an invalid login. To make sure it is invalid, just pick a way too uncommon username like *honorificabilitudinitatibus*,
- calculate the statistical average ($\frac{\sum_{i=1}^{n} x_i}{n}$ for valid login and for invalid one $\frac{\sum_{i=1}^{n} y_i}{n}$, where classes X and Y represent the response times for each valid and invalid try respectively) of the response time for a valid and an invalid login,
- try to login with the username, that you want to learn about its existence, for quite a few times,

Table 1. *Execution costs*

Execution Case	A set of instructions	B set of instructions
First execution	30	50
Alternated execution	20	40
Consecutive executions	10	30

  measure the response time and calculate the statistical average ($\frac{\sum_{i=1}^{n} z_i}{n}$ where Z is the response time of each try),
- if the average response time for the username you tried is closer to the average of valid logins, it is almost sure (as honest as Statistics can be) that the username you tried is valid too. If it's closer to the average of the invalids then probably it is invalid. If the number is by far different from the other two, then maybe the user is classified in another category (e.g. expired account) or an external factor altered your results.

External factors could be the traffic that a background process may produce, sudden loss of bandwidth or packet loss over the wire, line latency or even excessive CPU load. To eliminate the show up of external factors like these, make sure you use low-latency line, keep the same computer load and kill unnecessary processes running in background.

Evaluating Krahmer's work on EPTA

EPTA of Unix daemons, as described by Sebastian Krahmer, can be regarded as of extraordinary value in the sense that it revealed many attack vectors against computers. In simple words, it explains the fact that whatever we could do, it would take place in the arrow of time and therefore it can be traced from time related events. It has been one of the first attempts to explain how to fingerprint remote file system structures by measuring the times of the interrelated events they cause while in execution. Undoubtedly, this paper established the state of the art in execution path timing analysis of UNIX daemons.

Attacking a custom PAM service

It is time for theory to meet practise. To cover the practical part of this article the authors decided to perform a timing attack against the widely used ftp server software, ProFTPD 1.3.0. Earlier versions of this daemon suffered from timing leaks so its developers introduced a new module, called `mod_delay`, to secure the server from such kind of attacks. However, when we configured the daemon for our tests, we disabled this module for the concept to take place properly; we are sure there are still older and vulnerable versions of the server out there.

For the purposes of demonstrating the attack, a prototype tool, named `timat`, was coded. It is an implementation of the list of steps described in *What is a timing attack*. The function that executes the important instructions is `calc_time()` (see listing 1).

At lines 3 and 4, we create two references to the structures `timeval` and `timezone` since we need to keep two time values. The first value `tvalue1` holds the initial timestamp (just before we probe for the username) and the second one, `tvalue2`, holds the final timestamp (just after the FTP server responds with *Password*: at line 11). The `gettimeofday()` calls at lines 7 and 12 are responsible for saving the 2 timestamp values. The function `calc_time()` returns (line 13) the difference between the two values in microseconds. The

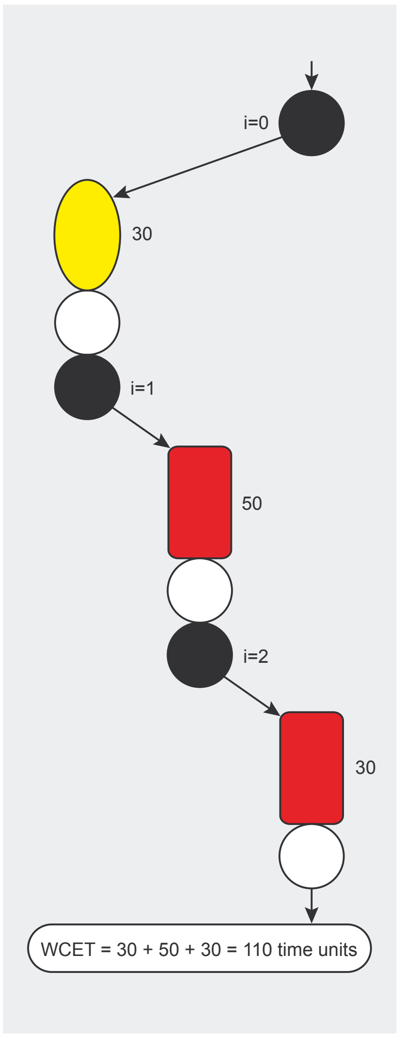


Figure 4. *Time analysis of the auth-mechanism*

Listing 2. *The decision maker.*

```
$ValidUser_Avg = check_user($valid_user, $host);
$GuessUser_Avg = check_user($check_user, $host);
$Factor = $ValidUser_Avg/$GuessUser_Avg;
if($Factor > 1.2)
{
 print "[+] User ", $check_user," does not exist!\n";
}
else
{
 print "[+] User ", $check_user," exists!\n";
}
```

Listing 3. *The implementation of forget()*

```
void forget()
{
 unsigned int time_slice;
 srand( time(time_t *)NULL) );
 time_slice = rand() % 31337 + 1;
 usleep(time_slice);
 return;
}
```

Note 1

The server was running on a default Slackware Linux 10.1 installation (Kernel version: 2.4.29) using an Intel Pentium II 334Mhz with 512Kb cache and 256MB of RAM. Network connection was a standard 10Mbit Ethernet in a LAN environment.

third argument the tool receives defines how many times the function `calc _ time()` should be called from within a for-loop. Remember, we need to probe for the username quite a few times in order to have some meaningful values and calculate the statistical average.

At that point, the only thing we have in hand is just a tool that calculates this average time value. How is this value going to helps us to discover whether a user exists or not? Assumptions can be made by comparing the average time of a valid user and the time of an invalid (binary classification problem), but this is quite time-consuming especially when we want to discover a lot of users. To simplify this procedure, a Perl script named *pr0ber.pl* has been coded to make the assumptions for us.

`pr0ber` uses the aforementioned `timat` tool to calculate the average times. It probes for a default valid user (root should be ok) and for a user, the existence of which is unknown. After getting the time averages from `timat`, it calculates the factor. ProFTPD 1.3.0 requires less time to handle an invalid user rather than a valid so we know that if the factor is greater than 1 (*GuessUser_Average<ValidUser_Average*) then the user we probed is definitely invalid. To minimize the chances of a faulty assumption, we increased the threshold of the factor to 1.2 (which returned 100% success during testing). A part of the Perl code, the one that takes the wild guess is shown in listing 2.

The `check _ user()` subroutine uses `timat` to get the average time for every user, while `$valid _ user` is set to `root` and `$check _ user` is the first argument of the script. To see the tools in action, have a look at *Attack results* later on.

Methods to measure time periods

I recommend you to take care of the minutes, for hours will take care of themselves, Philip Dormer Stanhope – *4th Earl of Chesterfield*, *Letters to His Son.*

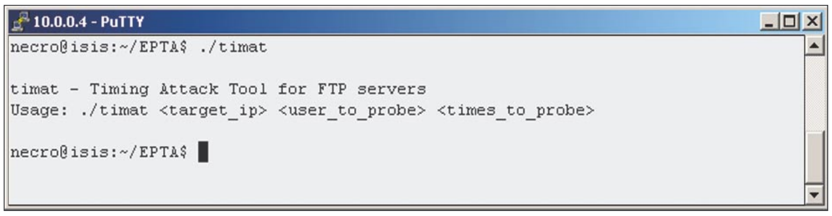
Generally there are at least two possible ways to measure how much time certain instructions require for their execution. The first one is the function `gettimeofday()`. It provides great flexibility and accuracy at the level of microseconds, thus describing the delay in a quite precise way where humans can hardly notice. It is as simple as keeping two time values returned from `gettimeofday()` and then calculating their difference.

The second one is using time-ticks. Sebastian Krahmer, in his paper (EPTA of UNIX daemons), refers to time-ticks as *number of calls to* `read()` *until reply is read.* To get a more clear view, what follows is a code-part of Sebastian Krahmer's patch for OpenSSH that uses time-ticks (instead of `gettimeofday()`)to calculate the delay.

```
while (read(peer, dummy,
sizeof(dummy)) < 0)
{
++reads;
}
return reads;
```

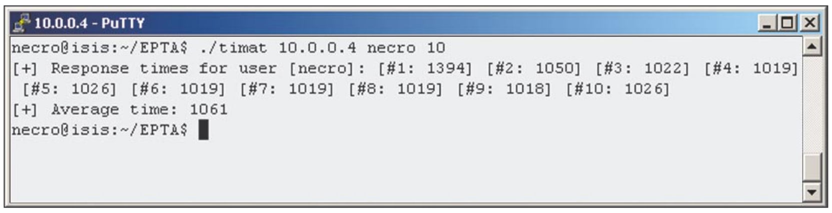
`reads` is an integer variable that constantly increases until the `read()` function is set to 1, which means that we are prompted to enter a password. Obviously, an invalid user produces different number of reads than a valid, so this can be considered as a competitive alternative measure.

In conclusion, Krahmer is way too comprehensive in his article so there is nothing more to add in this section, kindly described by him as *Choosing the right clock* in his paper.



```
necro@isis:~/EPTA$ ./timat

timat - Timing Attack Tool for FTP servers
Usage: ./timat <target_ip> <user_to_probe> <times_to_probe>

necro@isis:~/EPTA$ █
```

Figure 5. *How to use timat*



```
necro@isis:~/EPTA$ ./timat 10.0.0.4 necro 10
[+] Response times for user [necro]: [#1: 1394] [#2: 1050] [#3: 1022] [#4: 1019]
 [#5: 1026] [#6: 1019] [#7: 1019] [#8: 1019] [#9: 1018] [#10: 1026]
[+] Average time: 1061
necro@isis:~/EPTA$ █
```

Figure 6. *Probing for user necro (valid)*

Table 2. *Results as in figure 10*

User to guess	Class of user	Times probed	Accuracy of result
necro	Valid	10	100%
honorificabilitudinitatibus	Invalid	10	100%
root	Valid (super user)	10	100%
hakin9	Invalid	10	100%

Attack results

As described in figure 5, *timat* requires three input parameters. The first is the IP address of the host in mind, which in our tests had been set to `10.0.0.4`; the second is a user to probe and finally an unsigned integer number to define the number of probes.

Figures 6-8 show the tool in action, probing users `necro`, `honorificabilitudinitatibus` and `root` which are valid, invalid and valid users respectively. It's clear that invalid users require the least time to be handled rather than the super-user and a valid user which take a little bit more, but with a significant and visible difference. Note that our time measurements are in microseconds (1 microsecond = 1 × 10-6 seconds).

Figure 8 presents our *wrapper* tool, `pr0ber.pl` (see *Attacking a custom PAM Service*), which implements the decision making process regarding a user's existence. The first parameter of `pr0ber` is the user we want to guess about. Figure 10 displays the tool in action for various valid and invalid users.

As you can see in table 2, there is 100% accuracy for all users, which is an excellent performance if you consider that we make the assumptions based on statistics. In fact, even for less than 10 times of probing, the accuracy of the results had remained 100%. However in real-case scenarios, out of the testing environment, things might seem different. The two things that will guarantee accurate results, in that case, are greater samples for each probing (e.g. more than 10 per user) and an efficient calculation of the factor threshold in *pr0ber.pl*.

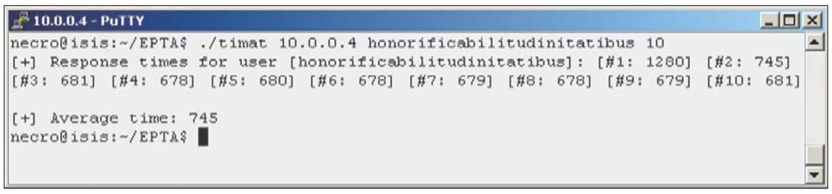
Counter-measures

Timing attacks are easy to perform, but it is also quite viable to protect against them. In general, there are two ways to do that. Whilst the first is more theoretical, the other is much more practical.

The theoretical way has the disadvantage of affecting the overall performance of the application and is more difficult to implement. The main concept is that the code that handles the authentication and the statements, that affect the flow, should be a completely balanced tree. This would ensure to a point that the program will respond in the same time for every class of input. However this 1-1 balance requires programming skills that nobody has.

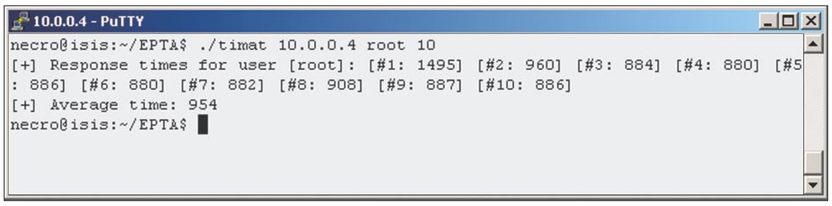
Random delays

Introducing random delays as *time-patches* is another trick that produces the same results. You implement the authentication part the way you would normally do. Afterwards, you calculate the exact response time for every class of login (see *What is a timing attack*). You keep the highest response time as an upper bound and you force the subroutines that handle the other classes of input to wait until they reach this upper bound. This can be easily done using the `usleep()` function of libc, although the hardest part is to calculate the



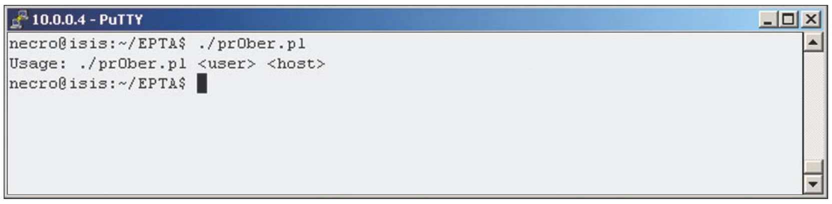
```
necro@isis:~/EPTA$ ./timat 10.0.0.4 honorificabilitudinitatibus 10
[+] Response times for user [honorificabilitudinitatibus]: [#1: 1280] [#2: 745]
[#3: 681] [#4: 678] [#5: 680] [#6: 678] [#7: 679] [#8: 678] [#9: 679] [#10: 681]

[+] Average time: 745
necro@isis:~/EPTA$
```

Figure 7. *Probing for user honorificabilitudinitatibus (invalid)*



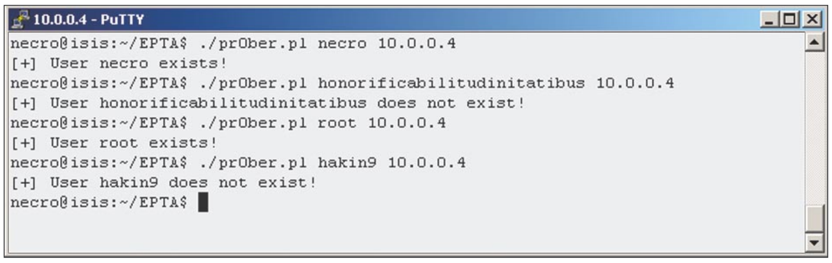
```
necro@isis:~/EPTA$ ./timat 10.0.0.4 root 10
[+] Response times for user [root]: [#1: 1495] [#2: 960] [#3: 884] [#4: 880] [#5
: 886] [#6: 880] [#7: 882] [#8: 908] [#9: 887] [#10: 886]
[+] Average time: 954
necro@isis:~/EPTA$
```

Figure 8. *Probing for super-user root (obviously valid)*



```
necro@isis:~/EPTA$ ./pr0ber.pl
Usage: ./pr0ber.pl <user> <host>
necro@isis:~/EPTA$
```

Figure 9. *How to use pr0ber*



```
necro@isis:~/EPTA$ ./pr0ber.pl necro 10.0.0.4
[+] User necro exists!
necro@isis:~/EPTA$ ./pr0ber.pl honorificabilitudinitatibus 10.0.0.4
[+] User honorificabilitudinitatibus does not exist!
necro@isis:~/EPTA$ ./pr0ber.pl root 10.0.0.4
[+] User root exists!
necro@isis:~/EPTA$ ./pr0ber.pl hakin9 10.0.0.4
[+] User hakin9 does not exist!
necro@isis:~/EPTA$
```

Figure 10. *Guesses for users necro, honorificabilitudinitatibus, root and hakin9*

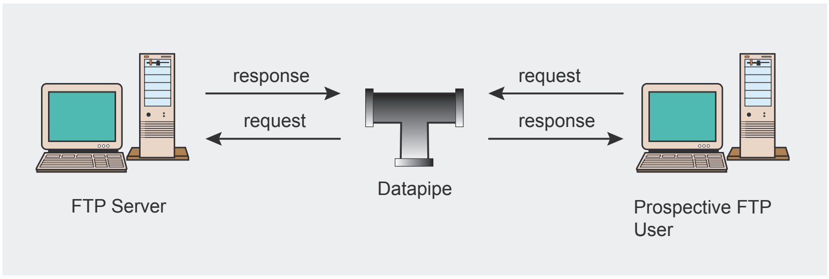


Figure 11. *A data pipe between a service and a user*

Note 2

The data pipe model introduces an overhead, that of the execution of its own instructions. Though on a theoretical basis, this overhead takes place in constant time and thus it is not an issue.

On the Net

- *http://en.wikipedia.org* – Wikipedia,
- *http://packetstormsecurity.org/groups/teso/epta.tgz* – EPTA of UNIX Daemons ,
- *http://www.proftpd.org* – ProFTPD,
- *http://www.openssh.org* – OpenSSH,
- *http://www.kernel.org/pub/linux/libs/pam/* – Linux-PAM,
- *http://www.grsecurity.net* – GrSecurity.

response times precisely. Result? The one desired. Every class of input requires the same time to elapse.

The practical way is quite similar to the theoretical one but it is more rough and abstract. The idea is simple. You introduce a pseudo-random and affordable delay before each response, making it impossible for an attacker to guess the *time-patches*. The seed pool of the random number generator can adopt values from the `gettimeofday()` function of libc. Alternatively, `/dev/urandom` will gather environmental noise from device drivers and create a fair good entropy pool to create random numbers from. The random number produced will be used as the parameter for `usleep()` to create the delay. Keep in mind that sleeping for much time will dramatically affect the performance of the application. If you plan to use PAM and you do not trust your imagination to produce random delays, you can use `pam_fail_delay()` function implemented in `security/pam_modules.h`. Make sure you make use of this function before every reply of your program or else timing attacks will be still possible at least when trying to obtain valid users. For your convenience, kernel security patches like GrSecurity are sophisticated enough to include security mechanisms for authentications.

Forgetful data pipes

Taking the aforementioned method of random delays into account, one may come up with many different architectural prototypes. One of them could include the network model of a data pipe.

A data pipe is a program which resides in the middle of a user and a service, like in Figure 11. Its role is to forward data received from the user straight to the service and vice versa. It therefore plays an intermediate role and thus it is able to control the relative data transmission timings (hint!).

Although the data pipe could be installed on a third-party computer, it is suggested that it should be running on the same computer as the service under the following strict policy. The ftp server is redirected to a port other than 21 and that port has to be filtered by a firewall so that it is completely unseen from the internet. The data pipe should run on the original service port (imitating its behaviour) and must have access to both the internet and the service.

Obviously, all prospective users ignore its existence and think that they communicate directly with the service. The key thing of this concept is to delay the final response back to the user so that the whole session can not be subject to accurate timing analysis. This can be done using the function in listing 3.

This function should be called just before sending the response back to the user.

Conclusions and further remarks

EPTA is a valuable technique that assists many developers to come up with optimal solutions. It is also possible to assist the dark side, as someone could use it to reverse engineer an artefact up to a degree (e.g. ranging from valid user exposition to total compromise of an asymmetric cryptographic system). In order to defeat and overcome such attacks, correct programming skills should be of our concern.

Einstein mentioned that time can be a fourth dimension, a dimension with different properties than these of space, obviously. Time is an illusion, something totally thought up of human beings just because it is the easy way to identify changes in our visibly spatial world. Illusion or not, we can safely support that this invisible entity contains much descriptive information about events turning it out to be a useful companion of our reality. ●

About the authors

Stavros Lekkas, originally from Greece, is an MPhil student at The University of Manchester (formerly known as UMIST). His interests include cryptography & information security, data mining, mathematics (logic, number theory and linear algebra) and computational complexity. He is currently working on his thesis which regards *Evolving Intelligent Intrusion Detection Systems*.

Thanos Theodorides, also from Greece, is studying Computer, Networks and Telecommunications Engineering at the University of Thessaly, Greece. A computer security enthusiast since his early teens, his interests include web development and security, wireless networks and network operating systems, among others. During his spare time he enjoys creating digital artwork.