

Execution path timing analysis of UNIX daemons

Sebastian Krahmer *krahmer@suse.de*

April 15, 2002

Abstract

Much effort has been taken over the years to give attackers as less information as possible when they map vulnerabilities. This includes the list of users which have access to particular machines or networks. For this reason even if remote access systems detect invalid usernames they continue the authentication process to not reveal the information to the attacker that he just issued an invalid login. This paper describes how one may get around these restrictions and still obtains valid usernames.

1 Introduction

Timing analysis has been done against a lot of things such as asymmetric crypto systems [timing], SSH traffic [Openwall] or against chipcards. Timing of execution paths in certain UNIX daemons or libraries has not been done or at least I am not aware of it. When attackers do timing analysis they hope to gather important information such as whether certain parts of supplied passwords are correct or, in our case, whether supplied usernames are correct. As much as low level CPU skills are needed for timing or power-analyzing chipcards we need programming and auditing skills to time execution paths of daemons or parts of programs and libraries. I ran across this problem when tracking down some strange SIGSEGV in a login program which turned out to use PAM in a wrong way. During the audit a vague idea was born and it turned out to be a very useful idea and hard to protect against, as the following sections describe.

2 Execution path analysis

Programs are deterministic automats which return the same result when fed with the same input. They also should take about the same time to compute their result when they are running in the same environment. Programs of our interest are authentication services such as PAM or SSH. For such programs there are following classes of logins:

- valid login
User is allowed to login. If i.e. a password is supplied correctly, the shell is executed.
- valid login with restrictions
The user exists but is not allowed to log in. For example because his account expired or he is listed in a deny-file.
- invalid login
The user does not exist but he is still prompted for a password. This does not show the attacker that an account name is invalid.
- special login
Administrator accounts e.g. root. Special actions are taken to ensure the login comes over a secure terminal etc.

The daemon handling the request executes different flows of control when handling the input, regarding to which class of login the username belongs. Additional syslogging for example may take place for invalid usernames. In surprisingly lot of cases we are able to distinguish between all four cases e.g. we can say to which class of login an arbitrary choosen username belongs by contacting the service remotely and measuring particular request/reply pairs. If we are not able to distinguish between all four cases we are often still able to distinguish between valid and invalid logins, which in most cases is important information for us. Other things attackers may get to know include knowledge about distributed authentication (NIS+ etc.), syslogging, or in case of some SSH implementations, whether users have an `authorized_keys` file. The worst case that may happen to a service is, that attackers are able to reconstruct the complete execution path e.g. which `if()` conditions have been triggered and how often certain loops have been executed.

2.1 The first real example: PAM

It is now time to look at a small example and explain why execution path analysis is important. The Pluggable Authentication Modules (PAM) are a framework to make authentication on UNIX systems as modular and flexible as possible. Almost all newer systems are PAMified and there are almost no services which do not use PAM. Today, if you contact an FTP, TELNET or SSH server, you can be sure that PAM handles the authentication process.

The following program is an example of using PAM. It uses the same authentication methods as the `ftpd` would use it, which probably uses the `pam_unix` module ¹.

```

1 #include <stdio.h>
2 #include <security/pam_appl.h>
3 #include <security/pam_misc.h>

4 struct pam_conv pc = { misc_conv, NULL};

5 int main()
6 {
7     pam_handle_t *ph;
8     int ret;
9     char buf[100] = "", *nl;

10    printf("PAM server 1.0 ready\r\n\r\n");
11    fflush(stdout);

12    fgets(buf, sizeof(buf)-1, stdin);

13    if ((nl = strchr(buf, '\n'))
14        *nl = 0;

15    ret = pam_start("ftp", buf, &pc, &ph);
16    if (ret != PAM_SUCCESS) {
17        printf("No!\n");
18        return 1;
19    }
20    pam_fail_delay(ph, 2000000);
21
22    if (pam_authenticate(ph, 0) == PAM_SUCCESS)

```

¹`pam_unix` does the standard password authentication known from UNIX

```

23     printf("Yep!\n");
24     else
25     printf("No!\n");

26     return 0;
27 }

```

To protect against timing analysis, PAM offers a special function: `pam_fail_delay()`. As we will see later, `pam_fail_delay()` fails to protect PAM against timing attacks, at least when attackers want to obtain valid usernames.

Within the PAM framework, the `pam_unix` module offers an authentication function which actually does the authentication job. In this function the `_unix_blankpasswd()` function is called for the supplied username.

```

...
1 /* if this user does not have a password... */

2 if (_unix_blankpasswd(ctrl, name)) {
3     D(("user '%s' has blank passwd", name));
4     name = NULL;
5     retval = PAM_SUCCESS;
6     AUTH_RETURN
7 }
8 /* get this user's authentication token */
...

```

This function tries to obtain a `passwd` structure for this user with the `getpwnam()` `libc`-function as seen in line 16. At line 17 it breaks, because now we have two possible execution-paths. The user may exist or it does not. If the user exists, additional code is executed. That includes using the shadow-system, dropping and regaining privileges and so on. This is measurable even if these few functioncalls seem to be too fast at first. It turned out that in `pam_unix` module for valid users about a dozen of additional syscalls are made plus a couple of calls to the `strdup()` and `memcpy()` functions. This is enough to decide between valid and invalid loginnames.

Of course you may have bad luck and the more on code you executed is lost 100 lines later because the other case is triggering similar conditions. This however should be very rare because it is unlikely that all classes of input take about the same time to be executed.

```

1 int _unix_blankpasswd(unsigned int ctrl, const char *name)
2 {
3     struct passwd *pwd = NULL;
4     struct spwd *spwdent = NULL;
5     char *salt = NULL;
6     int retval;

7     D(("called"));

8     /*
9     * This function does not have to be too smart if something goes
10    * wrong, return FALSE and let this case to be treated somewhere
11    * else (CG)

```

```

12     */
13     if (on(UNIX__NONULL, ctrl))
14         return 0;    /* will fail but don't let on yet */

15     /* UNIX passwords area */
16     pwd = getpwnam(name);    /* Get password file entry... */

17     if (pwd != NULL) {
18         if (strcmp( pwd->pw_passwd, "*NP*" ) == 0)
19             { /* NIS+ */
20                 uid_t save_euid, save_uid;
21
22                 save_euid = geteuid();
23                 save_uid = getuid();
24                 if (save_uid == pwd->pw_uid)
25                     setreuid( save_euid, save_uid );
26                 else {
27                     setreuid( 0, -1 );
28                     if (setreuid( -1, pwd->pw_uid ) == -1) {
29                         setreuid( -1, 0 );
30                         setreuid( 0, -1 );
31                         if(setreuid( -1, pwd->pw_uid ) == -1)
32                             /* Will fail elsewhere. */
33                             return 0;
34                     }
35                 }
36
37                 spwdent = getsppnam( name );
38                 if (save_uid == pwd->pw_uid)
39                     setreuid( save_uid, save_euid );
40                 else {
41                     if (setreuid( -1, 0 ) == -1)
42                         setreuid( save_uid, -1 );
43                     setreuid( -1, save_euid );
44                 }
45             } else if (strcmp(pwd->pw_passwd, "x") == 0) {
46                 /*
47                  * ...and shadow password file entry for this user,
48                  * if shadowing is enabled
49                  */
50                 spwdent = getsppnam(name);
51             }
52             if (spwdent)
53                 salt = x_strdup(spwdent->sp_pwdp);
54             else
55                 salt = x_strdup(pwd->pw_passwd);
56         }
57         /* Does this user have a password? */
58         if (salt == NULL) {
59             retval = 0;
60         } else {
61             if (strlen(salt) == 0)
62                 retval = 1;

```

```

63         else
64             retval = 0;
65     }

66     /* tidy up */

67     if (salt)
68         _pam_delete(salt);

69     return retval;
70 }

```

A small test-program which counts how often it is calling `read()` on the nonblocking socket before getting the password prompt suffices to show good results against the PAM server running via *inetd*.

```

lydia:root # ./a.out lp liane
[ 747 809 739 749 739 742 739 745 739 750 ]
-> 749.800000
lydia:root # ./a.out yard liane
[ 751 740 739 739 740 741 741 738 738 745 ]
-> 741.200000
lydia:root # ./a.out stealth liane
[ 713 716 716 717 716 720 718 719 717 719 ]
-> 717.100000
lydia:root # ./a.out invalid liane
[ 695 698 694 694 697 694 697 693 694 695 ]
-> 695.100000
lydia:root # ./a.out invalid2 liane
[ 693 692 695 694 694 692 692 694 693 756 ]
-> 699.500000
lydia:root #

```

The program counts the ticks² ten times and computes the average of these. The first two users *lp* and *yard* are special valid users listed in `/etc/ftpusers`³, which causes the *pam_listfile*⁴ module to execute additional code i.e. syslogging. *stealth* is just a normal login, and the last two are invalid. The timing was done against a 350 Mhz PII chip having a normal Linux setup and a 10Mbit ethernet connection to the timing computer which has a Pentium chip with 90Mhz. As said before we are able to distinguish between the classes of logins. The largest amount of code is executed for valid restricted logins, and much less code for invalid logins. The reason for this was shown above in the code snipped from the *pam_unix* module. Tracing the syscalls of the PAM-server shows that it indeed executes dozens of syscalls more in valid login-cases than in invalid login-cases.

As already noted PAM offers the `pam_fail_delay()` function to protect against the attack which just succeeded. From the manpage:

```
[...]
```

It is often possible to attack an authentication scheme by

²number of calls to `read()` until reply is read

³therefore are not allowed to login via *ftpd*

⁴FTP also uses *pam_listfile* to lookup deny-files

```
exploiting the time it takes the scheme to deny access to
an applicant user.
[...]
```

```
To minimize the effectiveness of such attacks, it is
desirable to introduce a random delay in a failed authen
tication process. Linux-PAM provides such a facility.
The delay occurs upon failure of the pam_authenticate(3)
and pam_chauthtok(3) functions. It occurs after all
authentication modules have been called, but before con
trol is returned to the service application.
[...]
```

The problem is that the delay only comes to play after all authentication modules have been passed. We however measured the time it took for the PAM conversation function ⁵ to reply with the "Password: " prompt after PAM was passed a username. This shows that the `pam_fail_delay()` function is useless.

2.2 A second example: OpenSSH

The same weakness which exists in PAM can be found too in OpenSSH up to the newest ⁶ versions. One example for different execution paths is at line 22. Only if the user exists `allowed_user()` is called ⁷. The called function obtains structs from the shadow file and checks whether the shell for this user is valid. OpenSSH sets a 'valid' flag in the `Authctxt` structure which allows later code to distinguish between valid and invalid users.

```
1 void
2 input_userauth_request(int type, int plen, void *ctxt)
3 {
4     Authctxt *authctxt = ctxt;
5     Authmethod *m = NULL;
6     char *user, *service, *method, *style = NULL;
7     int authenticated = 0;
8
9     if (authctxt == NULL)
10        fatal("input_userauth_request: no authctxt");
11
12    user = packet_get_string(NULL);
13    service = packet_get_string(NULL);
14    method = packet_get_string(NULL);
15    debug("userauth-request for user %s service %s method %s", user, service,
16    debug("attempt %d failures %d", authctxt->attempt, authctxt->failures);
17
18    if ((style = strchr(user, ':')) != NULL)
19        *style++ = 0;
20
21    if (authctxt->attempt++ == 0) {
22        /* setup auth context */
23        struct passwd *pw = NULL;
```

⁵The function that is responsible to talk to the user or the chipcard drive etc.

⁶3.0.2 as of writing

⁷due to lazy evaluation in the `if()` clause which is bad anyways

```

20     pw = getpwnam(user);
21     /* AUD: execution path :-) */
22     if (pw && allowed_user(pw) && strcmp(service, "ssh-connection")==0) {
23         authctxt->pw = pwcopypw);
24         authctxt->valid = 1;
25         debug2("input_userauth_request: setting up authctxt for %s", user);
26 #ifdef USE_PAM
27         start_pam(pw->pw_name);
28 #endif
29     } else {
30         log("input_userauth_request: illegal user %s", user);
31 #ifdef USE_PAM
32         start_pam("NOUSER");
33 #endif
34     }
35     setproctitle("%s", pw ? user : "unknown");
36     authctxt->user = xstrdup(user);
37     authctxt->service = xstrdup(service);
38     authctxt->style = style ? xstrdup(style) : NULL; /* currently unused
39 } else if (authctxt->valid) {
40     if (strcmp(user, authctxt->user) != 0 ||
41         strcmp(service, authctxt->service) != 0) {
42         log("input_userauth_request: mismatch: (%s,%s)!=(%s,%s)",
43             user, service, authctxt->user, authctxt->service);
44         authctxt->valid = 0;
45     }
46 }
47 /* reset state */
48 dispatch_set(SSHD2_MSG_USERAUTH_INFO_RESPONSE, &protocol_error);
49 authctxt->postponed = 0;
50 #ifdef BSD_AUTH
51     if (authctxt->as) {
52         auth_close(authctxt->as);
53         authctxt->as = NULL;
54     }
55 #endif

56     /* try to authenticate user */
57     m = authmethod_lookup(method);
58     if (m != NULL) {
59         debug2("input_userauth_request: try method %s", method);
60         authenticated = m->userauth(authctxt);
61     }
62     userauth_finish(authctxt, authenticated, method);

63     xfree(service);
64     xfree(user);
65     xfree(method);
66 }

```

At line 22 the if() is only evaluated once, when it gets its first user-auth-request packet. We have the chance to give invalid usernames more than once triggering a lot of different

execution paths. PAM usually also plays its role in the timing of OpenSSH servers but it is not possible to time, when the conversation function is called ⁸, because the password is sent along with the user and OpenSSH's conversation function just obtains the password from there. The `allowed_user()` function is just one example that it is possible to trigger different, measurable execution paths in OpenSSH. The hard thing about OpenSSH is that different versions behave different – timing wise.

Figure 1 shows a timing attack against an OpenSSH 2.2.0 server. It shows that this version ⁹ handles invalid logins faster than valid ones. Things are different for OpenSSH 3.0.2 for example ¹⁰:

```
lydia:# ./horstweg stealth liane
Probing for nobody [valid] ...
[ 0.010902 0.010327 0.010329 0.010280 0.010296 0.010384 0.010296 0.010285
 0.010415 0.010385 ]
avg: 0.0103265
Probing for h2o [invalid] ...
[ 0.011599 0.011127 0.011056 0.011073 0.011157 0.011082 0.012546 0.011080
 0.011064 0.011089 ]
avg: 0.011273125
Probing for stealth ...
[ 0.010901 0.010408 0.010320 0.010358 0.010283 0.010381 0.010359 0.010332
 0.010362 0.010292 ]
avg: 0.010350375
Factor valid/invalid is 0.916028164328879
Factor valid/guess is 0.997693320290328
Factor invalid/guess is 1.08915135925027
```

My guess is that 'stealth' is VALID

```
lydia:# ./horstweg zope liane
Probing for nobody [valid] ...
[ 0.013549 0.010292 0.010282 0.010328 0.010302 0.010301 0.010349 0.010296
 0.010307 0.010363 ]
avg: 0.010307125
Probing for h2o [invalid] ...
[ 0.011577 0.011081 0.011145 0.012325 0.011197 0.015682 0.011078 0.011076
 0.011134 0.011041 ]
avg: 0.01183975
Probing for zope ...
[ 0.010839 0.010329 0.010282 0.010286 0.010339 0.010263 0.010334 0.010313
 0.010320 0.010396 ]
avg: 0.01030825
Factor valid/invalid is 0.870552587681328
Factor valid/guess is 0.999890864113695
Factor invalid/guess is 1.14857031988941
```

My guess is that 'zope' is VALID

```
lydia:# ./horstweg zobel liane
Probing for nobody [valid] ...
[ 0.010341 0.010275 0.010327 0.010385 0.010292 0.010324 0.010296 0.010289
```

⁸as we did in the PAM example

⁹at least in my environment

¹⁰being indeed *zope* and *stealth* valid and *zobel* invalid

```

0.011460 0.010274 ]
avg: 0.010456
Probing for h2o [invalid] ...
[ 0.011622 0.011055 0.012410 0.011093 0.011053 0.011055 0.011113 0.010813
 0.011049 0.011108 ]
avg: 0.011205125
Probing for zobel ...
[ 0.011875 0.011094 0.011126 0.011070 0.011058 0.011117 0.011101 0.011036
 0.011122 0.011067 ]
avg: 0.0110905
Factor valid/invalid is 0.933144431677469
Factor valid/guess is 0.942788873360083
Factor invalid/guess is 1.0103354222082

```

```

My guess is that 'zobel' is INVALID
lydia:#

```

It tests one valid and one invalid account ¹¹ and looks whether valid/guess or invalid/guess is closer to 1. It does not matter, whether invalid logins take longer or shorter, they just need to be handled different. The patch may be downloaded from [epta].

3 Choosing the right clock

Finding a way to measure how much time certain actions of remotely located daemons require is probably the hardest part. Since we speak about microseconds, the delay is not visible for human beings. I tried `gettimeofday()` before and after receiving the reply and it looked like differences up to milliseconds are possible. I used 10Mbit networks and quite slow computers for testing. However timing attacks should be possible against fast computers too, if the machines are not connected to the net by modem but with a link with low latency. It turned out that using ticks is a good solution, too. To get the best results, following conditions should be met when using this approach:

- stop your downloads
- kill *cron* and *at* daemons on local machine
- do not run unnecessary programs in background and try to have the same load for every time-measure

For OpenSSH I included two patches, one using the `timeticks` just described and one using `gettimeofday()`, which is probably the better way.

I will try to get an appropriate lab environment with faster net and faster computers to see up to which point execution paths are measureable remotely.

4 Outlook

Due to the complex nature of today's daemons such as OpenSSH, a lot of additional information may be gathered. Since as showed even the underlying libraries play its role in the timing game and I would not doubt that gathering following infos is possible:

- the used *libc* version

¹¹this time using `gettimeofday()`

- whether remote site uses precompiled packages
- whether remote site is using certain patches for the service
- the order of how users are listed in `/etc/passwd`
- info about distributed authentication

Remote attackers may also fingerprint the system by probing for particular usernames such as *zope*, *wwwrun*, *gdm*, *fax*, *postfix* etc.. That way attackers get to know with high probability which OS/distribution in which version is running on the remote system because the defaultlist of valid usernames differs from distro to distro. If you did not care about this timing-attack before – now you should.

5 Counter-measures

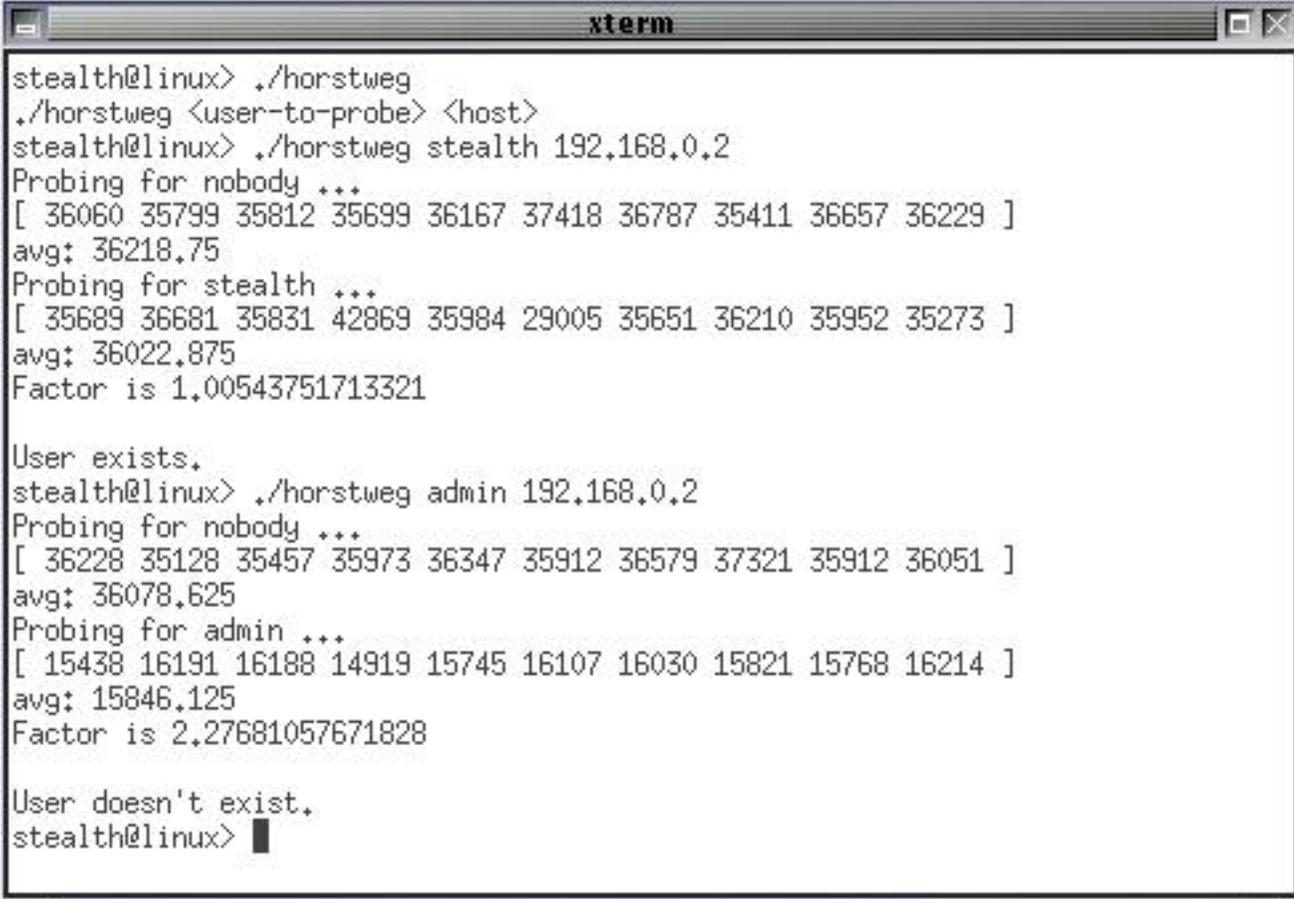
The only thing one could do, is putting the daemon to sleep for a random amount of time like `pam.fail_delay()` before every reply. Maybe it is even possible to stick the delay part in the TCP/IP stack usable via special socket-options but I think this would break RFCs. You probably do not want to have a balanced tree of `if()`-conditions in your programs. I think it is very hard to force your program to take about the same time for every class of input, except you measure that for certain input you have been too fast and wait the amount of microseconds to reach execution times other classes of input have.

Acknowledgments

Thanks to various people for proof-reading and discussion.

References

- [timing] Cryptography Research, Inc.
whitepapers about timing attacks and other interesting things
<http://www.cryptography.com/resources/whitepapers/>
- [Openwall] The Openwall team.
several advisories also covering timing issues
<http://www.openwall.com/advisories/>
- [epta] me ;-)
execution path timing analysis paper plus programs
<http://stealth.7350.org/epta.tgz>



```
stealth@linux> ./horstweg
./horstweg <user-to-probe> <host>
stealth@linux> ./horstweg stealth 192.168.0.2
Probing for nobody ...
[ 36060 35799 35812 35699 36167 37418 36787 35411 36657 36229 ]
avg: 36218.75
Probing for stealth ...
[ 35689 36681 35831 42869 35984 29005 35651 36210 35952 35273 ]
avg: 36022.875
Factor is 1.00543751713321

User exists.
stealth@linux> ./horstweg admin 192.168.0.2
Probing for nobody ...
[ 36228 35128 35457 35973 36347 35912 36579 37321 35912 36051 ]
avg: 36078.625
Probing for admin ...
[ 15438 16191 16188 14919 15745 16107 16030 15821 15768 16214 ]
avg: 15846.125
Factor is 2.27681057671828

User doesn't exist.
stealth@linux> █
```

Figure 1: Timing an OpenSSH 2.2.0 server.