New Side Channels Targeted at Passwords

Albert Tannous[†] Jonathan Trostle[‡] Mohamed Hassan[†] Stephen E. McLaughlin[†] Trent Jaeger[†]

[†] The Pennsylvania State University
University Park, PA 16802
{tannous,mhassan,smclaugh,tjaeger}@cse.psu.edu

[‡] Johns Hopkins University APL
Laurel, MD 20723
jonathan.trostle@jhuapl.edu

Abstract

Side channels are typically viewed as attacks that leak cryptographic keys during cryptographic algorithm processing, by observation of system side effects. In this paper, we present new side channels that leak password information during X Windows keyboard processing of password input. Keylogging is one approach for stealing passwords, but current keylogging techniques require special hardware or privileged processes. However, we have found that the unprivileged operation of modifying the user key mappings for X Windows clients enables a side channel sufficient for unprivileged processes *to steal that user's passwords, even enabling the attacker to gain root access via* `sudo`*. We successfully tested one version on Linux 2.6; we were able to obtain a high degree of control over the scheduler, and thus we can obtain accurate timing information. A second version (logon detection) works without depending on accurate clocks or cache effects. Thus, in addition to demonstrating new side channels, we show that (a) side channels cannot be eliminated by removing accurate clocks or hardware cache mechanisms (b) side channels are of continued concern for computer security as well as cryptographic processing.*

1. Introduction

Computer security has historically been focused on preventing untrusted programs from obtaining access to information that would violate security policy, where the policy is realized in the form of Mandatory Access Controls (MAC) [16]. Although Mandatory Access Controls are effective for control of the system designed communication interfaces, it has long been known that covert channels [6] can obviate this protection and lead to security policy violations. A covert channel is defined as a communication channel between two entities that does not use the system defined communication interfaces. For example, one process can manipulate the disk in order to send information to another process, where the security policy would prevent normal communication between these processes.

A related threat, usually in the context of cryptographic processing, is side channels. Here untrusted malicious programs are able to observe some aspects of a shared system and obtain information about cryptographic keys due to the side effects of cryptographic processing (the cryptographic process is trusted and does nothing to aid the malicious process in contrast to a covert channel).

With the exception of the Tenex flaw [1, 7], side channels are primarily the concern of cryptographic algorithm developers, and covert channels are of concern to computer security design and development. Thus side channels have largely not been a concern to computer security. In this paper, we present side channels that disclose information about user passwords, and thus we show that side channels continue to be of concern for computer security design. (In current commodity OS-based systems, there are more efficient attacks, but side channels are a concern for systems with more advanced security features).

1.1. Related Work

One of the earliest examples of a side channel is the Tenex flaw [1, 7] where passwords are vulnerable since they were checked one character at a time, and processing stopped at the first incorrect character. Thus an attacker could guess password characters based on the amount of time needed to process the password.

More recently, Kocher demonstrated timing attacks against RSA and other public key algorithms [5]. The papers [12, 11] also demonstrate cryptographic side channels against RSA, where [2, 9] demonstrate side channels against AES. Countermeasures against cache based channels are presented in [10, 17].

Trostle [14] presents a side channel against the Xlock program, using information associated with X Windows processing. [14] shows that keyboard interrupts and X processing can be detected and measured by an unprivileged process running on the system. Classical covert channel countermeasures [3, 15, 4] will have some effectiveness in limiting or closing side channels as well. In particular, [3] is aimed at removing accurate clocks which many channels depend on.

1.2. Our Results

In this paper, we present side channels that disclose information about user passwords. These channels leak password information during X Windows keyboard processing of password input. We demonstrate that it is possible for an *unprivileged process* to steal a user's password using the following sequence of steps: (1) remap that user's X Window key map to generate a measurable timing side channel for keyboard entry processing (2) that the effects of this side channel can be measured by an unprivileged process external to the victim; and (3) use these measurements to guess a password from an unprivileged process without access to the password file. We show that this side channel is an appropriate mechanism to extract passwords from X client programs (e.g., screensavers and command line programs). The key facet of this attack is that it can be executed by unprivileged processes, rather than requiring privileged process access of typical keyloggers, hardware-based attacks, such as JitterBugs [13], or unauthorized access to X Windows processing directly. Even if all of these approaches are prevented, this side channel may still be leveraged to obtain access to a user's password, including users who may have the privilege to use this password for `sudo` processing.

Our first attack is based on a *remap timing channel*, and we demonstrate an implementation of an unprivileged, multi-threaded attack process on a Linux 2.6 system. With the 2.6 scheduler, we are able to obtain a high degree of control over the scheduler, and thus we can obtain highly accurate timing information. In particular, we have implemented a multithreaded timing program which is able to accurately time run durations of other interactive processes. It it likely that this capability has other applications. Our second channel, although noisy, is effective on quiescent systems and is able to obtain enough information about an eight character password to make a subsequent guessing attack tractable. We have successfully tested it on a Linux 2.4 kernel system. Our second algorithm is ineffective on our 2.6 system. Both of these channels scale well as password length increases. We also present results for a 3rd channel: the logon detection channel. This channel simply detects whether a given character is in the password by determining whether an initial logon is successful or not. In other words, a given character is remapped, prior to password entry, and the character is not in the password if and only if the initial logon is successful. All attacks are performed using only unprivileged processes.

The basic mechanism for our side channels leverages the X Windows keyboard remapping capability (using the `xmodmap` command). We can remap keys or a subset of keys to a character that requires a longer time for X processing. Thus if the password contains the remapped key, then that will be detectable by an untrusted program based on the additional processing time. The user will be unable to detect that the key has been remapped, since the remapped key is not echoed to the terminal. However, if the password contains a remapped key, then the password will be invalid. The attacker must immediately remap the keys back to the pre-existing configuration so that the second logon succeeds. If this attack occurs infrequently, then it will not be detected by the user. Also, when the attack only remaps keys not contained in the password, then those runs will not cause login failures. This latter channel (the logon detection channel), has the advantage that it does not depend on the hardware cache and is also effective in the absence of accurate clocks. Thus countermeasures aimed at removing accurate clocks [3] or the hardware cache mechanism for side channels [10, 17] will not effect this channel. A limitation of the logon detection channel is that it does not give information about which password characters are in which positions; thus a follow-up guessing attack will require more time. It is unlikely to be effective against passwords longer than 9 characters, unless combined with some other attack.

As an example of the remap timing channel, suppose the attacker remaps the 'a' key and the user password contains one 'a' as well as other characters. Then the attacker program can time the X Windows processing for each password character. The attacker will see that one character requires a longer processing time and will conclude that this character in the password is the 'a' character. Both unprivileged user and the root passwords can be targeted with these attacks. This remap timing side channel is more efficient than the original Tenex flaw attack, since it samples one or more keys across multiple password characters during a single logon.

We have conducted experiments that validate these channels on the Linux operating system with X Windows. In principle, these attacks can also be carried out on other operating systems that support keyboard remapping, such as the Windows operating system.

1.3. Organization

The paper is organized as follows: Section 2 covers some preliminaries including X Windows background. Section 3 overviews our remapping side channels. In Section 4, we show that the remap timing channel exists and can be used to determine passwords from an unprivileged process on Linux 2.6. We also briefly examine this channel on a system with a different scheduler, Linux 2.4. In Section 5, we present the logon detection channel which leverages the same remapping mechanism to identify password characters based on login failures. Section 6 covers guessing attacks, as we do not have access to the password file. In Section 7, we briefly discuss countermeasures. In Section 8, we briefly discuss the impact of user input errors and other issues. We conclude in Section 9.

2. Side Channels in X Windows

We now briefly examine X Window keyboard input processing. In the X Windows system, the X server process receives mouse and keyboard input interrupts from the operating system. The X server then sends these X events to interested X clients which process the events. To enable alternative keyboard layouts, the `xmodmap` utility allows (unprivileged) users to remap the keyboard. Thus the mapping between keys and characters can be changed. There are three mappings that occur between the time a key is pressed, and the time a character is displayed on the screen in the X Windows system [8]. They are as follows:

1. **Physical keys to keycodes:** This translation is X server dependent, and client processes cannot detect this. We will not mention it any further.

2. **Keycodes to keysyms:** This mapping can be modified by the X clients themselves, but applies system wide. As we will see, the remapping utility (`xmodmap`) enables the side channels. The keysym is a logical entity which carries the meaning of a keypress. Examples of keysyms include XK Return and XK Space, which represent the return key and the space bar respectively.

3. **Keysyms to strings:** A keysym itself contains no information about whether or not a character should be displayed for a given keypress or how. It is up to the client process to work with the X server to perform the keysym to string mapping, where the string is zero or more characters to be printed for a given keypress. This translation is performed in the X client by the X library function *XLookupString*. We noticed that the *XLookupString* has two code paths for different types of symbols. One path does translations for ascii characters and the other for unicode characters. The path for unicode characters is longer, but also importantly, not leveraged for normal password characters. This unicode code path in *XLookupString* will form the basis for our first two timing channels.

The combination of our ability to remap the keycode to keysym mapping in the X server and the presence of this extended code path in *XLookupString* provides the basis for a timing channel that an adversary could leverage. An unprivileged process, under the control of an adversary, may remap a keycode using `xmodmap` to a keysym that corresponds to a unicode character, thus resulting in the execution of this extended code path. Further, since passwords almost always consist of ascii characters only (at least in the US), the execution of this code path will be infrequent, so a significant instruction cache impact will be likely.

As a result, if the adversary can setup the system to cause such overheads and effectively measure the delays inherent to such overheads, then the adversary can detect when a victim pressed a key that has been remapped, enabling prediction of the key. Designing and implementing an attack approach that enables these functions is non-trivial as we describe in the following section.

3. Remapping Side Channels

In this section, we overview remapping and its application to side channels. At a high level, the adversary's goal is to learn information about characters in a secret string (e.g., password) by remapping some keys on the keyboard. Depending on the X client and channel, the adversary must determine: (1) what is a remapping that will enable the execution of the extended code path in *XLookupString*, (2) when to initially remap the keyboard (prior to password entry), (3) how the measure the channel, (4) when to map the keyboard back (after the password has been entered), (5) how to determine the password from the measurements. These tasks are challenging; remap and map back must occur at the right times else the attack will fail or be discovered. Measuring is also potentially challenging, depending on the channel.

For the remap timing channel, our strategy is to remap a subset of keys to a character that requires a noticeably longer time for X Windows processing. The third X Windows mapping described above, keysyms to strings, is relevant here. We remap the selected keys to the **euro**: 0x20ac (hexadecimal), since the **euro** exercises the extended code path described above (actually, the **euro** is deep in this path). Also, the **euro** is unlikely to be a password character.

In addition, the first time the unicode code path is taken a significant number of instruction cache misses result, further increasing the processing time. Our experiments validate this hypothesis; roughly 10000-20000 additional cycles are needed to map the **euro** keysym the first time.

The 3rd X mapping occurs in the X client. Thus if we are able to time the X client processing, then we will likely notice whether the associated key has been remapped. Our strategy is to run a timing process (with one or more threads) both prior to the X client, and after the X client.

In the first channel below (Linux 2.6), the timing process is able to time both the X server and the X client separately (see Figure 2). This results in accurate measurements, since the X client processing duration is what we want to measure. In the second channel described below (Linux 2.4), the timing process will time both the X server and the X client (see Figure 1).

We call the other remapping channel mentioned above, that detects whether logon is successful or not, the logon detection channel. This channel is noiseless, and if used alone, can narrow the password space to the set of characters contained in the password.

4. Remap Timing Channels

We present the remapping timing side channels in this section. First, we describe an initial experiment with one process that failed to detect the remapping channel. We then performed a second, detailed timing experiment that confirmed the existence of the remapping channel. We find that the remapping channel is caused by a combination of longer code path to process unicode and the overhead due to instruction cache misses when this code path is taken. In the third experiment, we show that a multi-threaded attack process can successfully measure the remapping side channel. The use of a multi-threaded attack process enables us to maintain the scheduling priority of the attack processing, so we can ensure that it runs directly prior to and after the X client victim. Our attack depends on the Linux 2.6 scheduling algorithm. We examine the attack on Linux 2.4 to discuss the impact of a different scheduler.

The machine used for the Linux 2.6 experiments has a Intel Pentium M-740 (1.73GHz) processor and runs the Ubuntu 6.10 operating system (Linux Kernel version 2.6.17-11). Our 2.6 system is a wireless laptop.

We ran experiments using a generic X client that accepts keyboard entry like a program for entering a password, such as `su`, `login`, or the SSH client. It does not include all of the display functions of screen locking programs, such as `xscreensaver`, although the amount of display update activity is not great when a password is being entered. Our X client processes user keyboard events that are sent by the X server. It works by enabling the user to type a string (e.g., a password), and it prints out the received character string after receiving a return character.

4.1. Single Process Experiment

In this section, we show that a single process is unable to detect remapped keys due to noise on the system. Our initial (Linux 2.6) experiment used a single timing process that detects and measures other process activity by sampling a timer in a loop. Thus when the timing process detects a large delay between consecutive timer samples, it can reliably associate this duration with other process activity. As discussed in Section 3, we remap one or more keys to the

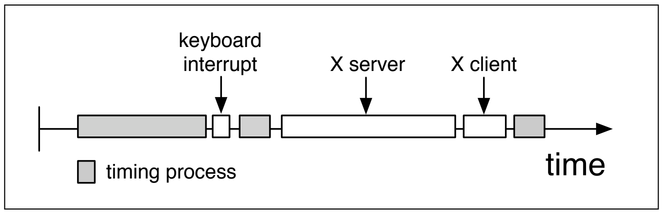

Figure 1. Scheduler Timeline with One Timing Process and X Windows Processing: a single timing process follows both X server and X client, resulting in noisy measurements.

euro keysym in order to detect the longer processing associated with these keys.

For our experiment, we initially needed to match known X processing events in the X client with the corresponding observed process activity in the timing process. Without this information, we would not know what types of process activity durations to look for. This task was easily accomplished using timestamps in both the X client and the timing process. (Note: in Section 4.3, where we perform the real experiments against passwords, we do not use timestamps in the X client. In other words, the X client is treated as a fixed program that cannot be modified by the adversary.) By matching the timestamps, we could clearly associate the X processing events with the durations observed in the timing process.

The first thing we observed was that the X server and X client ran consecutively (see Figure 1). In other words, our timing process did not run in between the X server and X client. We determined that this was because our timing process was CPU-intensive, so its scheduling priority was lowered, preventing it from being scheduled before the X client when it is ready. Other processes sometimes run in this X processing window as well, extending the duration between our timing measurements with unrelated processing. Further, this other processing also caused variability in X server processing times due to cache conflicts. Thus the amount of noise prevented us from reliably detecting remapped keys.

4.2. Confirming the Side Channel

We conducted a second experiment to determine if there is a detectable timing channel in X key processing. We did this in two steps. First, we measured `XLookupString` directly from within the X client. The idea is that if a timing channel exists, then the X client processing times for remapped keys, even measured within the X client, should be demonstrably longer. We found that it consistently takes over 15,000 additional cycles to process a remapped character. Non-remapped characers are processed in 5,000 to 7,000 cycles, while remapped ones would take over 25,000

	Run 1	Run 2	Run 3	Run 4	Run 5
t	17411	18530	13699	16581	16095
r	11917	12123	15643	10424	13821
y	11290	14653	10735	14674	12776
euro	37942	39439	37272	32001	37410
f	12901	12288	11615	15361	11746
i	12171	16296	15445	11777	17146
n	15950	11359	12209	12882	10830
d	12563	14813	17275	13609	14462

Table 1. Five runs containing the cycle counts for the X client key processing of `XLookupString` from an external, unprivileged process where semaphores are used to synchronize the processes. The euro character is the remapped character, and its processing takes at least 10,000 cycles more than any other character.

cycles.

Second, we verified that we could see this timing delay from an external process under idealized conditions. In this case, we have the X client and the attack program share a semaphore, such that the X client wakes the attack program to take timing measurements every time that the X client completes `XLookupString` processing. Table 1 shows the resultant times as measured by the timing program over eight runs. The fourth character in a password `try2find` (i.e., the '2') is mapped to the **euro** character. This table shows that the remapped character processing takes at least 10,000 cycles more than any other character. Further, the average difference is close to 20,000 cycles. Based on these experiments, we are encouraged that the timing channel in X client key processing can be measured by an unprivileged, external process.

4.3. Multithreaded Experiment

In this section, we show that a multithreaded timing process is able to detect remapped keys. The reason is that multiple threads are able to exploit the scheduler in a manner that allows the threads (or tasks) to maintain the highest interactive priority. Thus, a timing thread will run in between the X server and the X client and the X client CPU duration can be accurately measured.

The basic idea of this attack is as follows. We remap the keyboard prior to user password entry by determining a password entry program is being run (e.g., `su`) or that the screen is locked (e.g., for `xscreensaver`). Through maintaining several timing threads [1]at high priority, we are

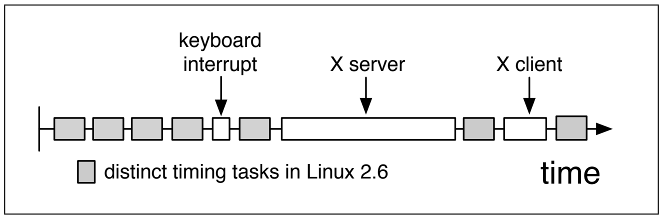

Figure 2. Scheduler Timeline with Multiple Timing Processes and X Windows Processing: timing processes run at same priority as X client victim, so they run both right before and right after X client.

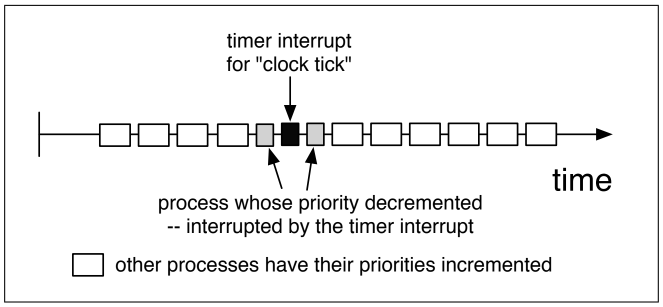
timer interrupt for "clock tick"
time
process whose priority decremented -- interrupted by the timer interrupt
other processes have their priorities incremented

Figure 3. Timing Process Priority Adjustments in Linux 2.6. Only the process that is interrupted by the timer interrupt has its priority decremented. Other processes see their priority incremented. Thus running a large number of timing processes results in each maintaining high priority.

able to run timing processes both immediately before and immediately after the X client. The processing timeline is shown in Figure 2.

In Linux 2.6, the scheduler rewards threads that sleep over one or more clock ticks. When a thread is awakened from sleeping, it receives a boost to its dynamic priority based on the amount of time that it has slept. Thus, if we run multiple timing threads, where the threads use semaphores to awaken each other in a synchronous manner, then most of the time a thread will be sleeping and thus receive a bonus vs. being decremented.

Thus, at any clock tick, one of the timing threads will be running, so its dynamic priority will be decremented. The other timing threads will be asleep, and they will have their dynamic priority increased, if an increase is possible (see Fig 3). (An increase will not be possible if the thread has already received the maximum bonus adjustment to its priority). Thus we see that if we run n timing threads, then on average, we expect a given thread to be eligible for a priority increment $\frac{n-1}{n}$ of the time, and to receive a priority decrement $\frac{1}{n}$ of the time.

[1]In Linux, `tasks` represent both processes and threads, so the scheduler treats threads as it would a process. We use the term threads to indicate that these Linux tasks share an address space, as threads in the same process would.

	Run 1	Run 2	Run 3	Run 4	Run 5
t	47147	42468	47223	42488	47572
r	31598	33740	32462	30385	43398
y	29026	39900	35825	26788	30639
euro	64316	53888	62909	59987	65099
f	31195	39115	32239	35956	28793
i	27953	33941	40367	32694	45323
n	40314	40005	28643	46316	33041
d	27704	32707	28644	29866	43798

Table 2. Timing the keypresses from the multithreaded attack program (timed in cycles).

In Linux 2.6, interactive threads receive a maximum bonus of -5 to their dynamic priority. If there are more than two timing threads, then their dynamic priorities will tend towards their static priority minus 5 (i.e., a lower priority is better for scheduling). The X client will also tend toward the same priority. Thus when the X client awakes and becomes runnable (due to input from the X server), it will be placed in a priority queue behind the existing timing threads. Therefore, a timing thread will run immediately prior to the X client, and another one will run after the X client. Thus, we should obtain an accurate indication of the remap status of the individual keys.

4.3.1 Attack Program Design

We describe the design for the Linux 2.6 attack program. The idea is to use multiple timing threads that time the victim X client's processing. These threads will use semaphores to hand-off responsibility for timing. The basic pseudo-code skeleton for each timing thread is as follows:

```
while() {  // timing loop
sem_wait(sem_id1);  // block awaiting
            // increment of semaphore
time1 = sample_the_timer();
if(time1 is in range of interest)
   record time1;
sem_post(sem_id2); // increment the
// semaphore for next timing thread
}
```

The results for the implementation of this approach are shown in Table 2. Here again, we entered the string "try2find", with the character '2' remapped. As we can see, the remapped character stands out very clearly. Regular characters are in the range of 30,000 to 45,000 cycles. The line for the **euro** character shows that the duration of the remapped characters ranges between 55,000 and 75,000 cycles.

X Server Duration	X Client Duration	Remap Status
228086	42068	Not remapped
157738	34192	Not remapped
169017	51962	Remapped
208754	37976	Not remapped
144332	27710	Not remapped
249284	42095	Not remapped
219884	45322	Not remapped

Table 3. Output Showing Remapped and Non-Remapped Keys (timed in cycles)

4.3.2 Solving the Noise Problem

In the above attack, the problem of extraneous processing exists. In particular, other activity can result in additional time durations that are in the same range as the timing measurements. Thus, we have a noisy channel where the main noise component consists of additional measurements. We have largely solved this problem by noting that the keyboard processing consists of the X server processing (around 150,000 to 250,000 cycles) immediately followed by the X client processing (approximately 30,000 to 80,000 cycles). The use of this signature helps to remove most of the noise from the channel that would be present if we only focused on the X client durations.

Table 3 shows a typical output sequence. The left column contains X server process durations (in cycles), and the second column contains X client process durations. These pairs of values are readily identifiable in the timed process durations, and our test results indicate that false positives (consisting of a pair of values in the same range) occur very rarely. The same is true for false negatives with the exception of the first character; occasionally we miss (a miss occurs when the X client processing doesn't immediately follow an X server processing time) the X client processing for this character since the X client's priority may not have fully recovered from it's initial processing when it handles the first password character. Succeeding characters receive the benefit of the extra sleep time which allows the X client's priority to be as high as the timing threads. Also, the 1st password character processing is more likely to suffer cache misses vs. later characters. One of the timing threads runs before, between, and after each pair of processing durations in the figure.

4.3.3 When to Map Back - Identification of Return Key

Password entry is often immediately followed by a return character. If the timing process is able to identify the return character, then it can remap the keyboard back to the original state at that point. In this way, the timing process will not be detected since subsequent keyboard output will echo in the normal manner.

Run 1	Run 2
181469 (X)	178566 (X)
120042	125654
198459	207160
128269	134359
95050	82294

Table 4. Return Key Signature, as seen in two typical runs starting with X server processing (X), measured in cycles

In our Linux 2.6 experiments, we have observed that our X client has a consistent signature for the return key processing. This is partially due to the fact that our X client prints to the terminal upon reading the return. Many X clients have similar behaviour upon successful logon. The signature for two distinct runs is shown in Table 4.

We see an initial X server processing duration immediately followed by several other times associated with X processing.

4.3.4 Test Results

Using the algorithm described above, we ran tests against a generic X client on a Linux 2.6 system. Each test consists of entering the same 8 character password. During the test, the timing program runs and we subsequently analyzed the results. We remapped a single key per test. Our results are given in Table 5. The amount of noise was minimal. Our results for the first password character were the most problematic, but otherwise the results are fairly accurate. In particular, if we throw out the first character as unreliable, then for 84 characters transmitted, we have one error (in the 2nd character), 4 not received characters (but we know that we have missed these particular characters), and 79 characters transmitted correctly, which is over a 90% success rate.

Upper case characters include both a shift key processing time in addition to the upper case key time. The upper case processing time is longer when the key is remapped. Although we tested upper case characters, we did not actually analyze any passwords that included upper case characters. Nevertheless, we believe such analysis would be straightforward.

We also ran tests over several (up to 6) minutes to examine the priorities of our timing threads. We confirmed (using the `ps` command) that the threads remained at the highest interactive priority throughout the interval. Thus our multi-threaded design remains priority stable.

4.3.5 Optimal Remapping Strategy

We also experimented with remapping multiple keys. When more than one password character is remapped, the 2nd remapped character in the password will usually not require as much additional processing time as the first character. The reason is that the additional code will already be in the cache. Also, modern processors will attempt to optimize and guess ahead regarding upcoming code paths. Thus our experiments did not show significant benefit to remapping more than one password character at a time. Given this constraint, there is still some advantage to remapping two keys per test, given the high probability that only one password character will be remapped per test. With this strategy, 35-40 logons are needed, on average, for an 8 character password given a 94 character alphabet. (It is sufficient to reduce 6 of the 8 password characters to two possibilities each, and guess the remaining two characters - see Section 6 for guessing attacks).

Test	Results
1	Missed 1st character
2	No errors
3	Missed 1st character
4	1st and 2nd characters appear remapped but aren't
5	No errors
6	7th character unknown (two closely spaced character times) missed 4th character (X server and X client ran consecutively)
7	No errors
8	missed 1st character, missed 8th character
9	1st character is incorrect, missed 2nd character
10	1st character is incorrect, rest are correct
11	1st character is incorrect, rest are correct
12	1st character is incorrect, rest are correct

Table 5. Test Results on Linux 2.6: Each Test Consists of Entering Same Password

4.4. Linux 2.4 Side Channel

Ideally, we would like to run timing processes or threads immediately before the X client and immediately after it as well. This approach gives the most accurate timing model. The Linux 2.4 scheduling algorithm makes this approach difficult; it is more difficult to exercise control over the scheduler than in Linux 2.6. The reason is that it is not possible for a Linux 2.4 task to run at frequent random times and still maintain a high dynamic priority (since it can't make up the priority decrements by sleeping as in Linux 2.6). We have settled on a model where a timing process runs before the X server and then after the X client (see Figure 1). Thus we measure the X client along with the X server. This model results in additional noise. Nevertheless, on a quiescent system (a standard Linux 2.4 desktop system without additional software packages or wireless networking), our experiments have resulted in narrowing the password space to the point where guessing attacks are feasible.

Due to the additional noise, we require more tests than in the Linux 2.6 case. Due to space constraints, we omit the details.

5. Logon Detection Channel

Here, we present a simpler, but less accurate, timing channel that leverages the same remapping mechanism. This channel measures login failures caused by remapping. When a key is remapped, we observe different behavior depending on whether the remapped key is in the password or not. This channel can be measured using much coarser timing, such as `timeofday`, but it also results in less reduction in the password space than for the remap timing channel.

The basic idea is to time the difference between start of password entry and successful logon. Successful logon, say for the `su` program running in an `xterm`, can be detected by the existence of a newly created root shell. The start of password entry can be detected by the fact that the `su` program is running. If a single key is remapped, then logon success is determined by whether this character is part of the typed string. Note that the time difference between successful logon (on first try) and successful logon (on second try) will be on the order of several seconds. Thus, this channel will not depend on accurate clocks.

For example, a timing program can remap one key per run. If we assume a password alphabet of 94 characters, then we will, on average, run about 86 times to reduce the password space to 3 bits or less per character (since we cannot determine the position of characters in the password using this attack). Then the remainder of the space can easily be searched by a guessing program, if we assume passwords of about 9 characters or less (we present our guessing program and its performance in Section 6). Thus a 9 character password would be compromised after approximately 86 logons, plus a small amount of time for guessing.

5.1. Logon Detection Channel Algorithm

Here, we give the details for the logon detection channel attack; our X client is `su` in an `xterm`.

We must make three decisions shown in Figure 4: (1) when to remap a key, designated as T_0; (2) when to initiate detection of successful logon, T_1; and (3) when to un-remap the key, T_2. The accuracy of the first decision determines whether we can capture the first character. The accuracy of the second decision determines whether we distinguish between success and failure correctly. The accuracy of the third decision determines whether we capture all characters (i.e., do not un-remap too soon) and avoid detection (i.e., do not un-remap too late).

For `su`, we determine that it is being run by checking for the process in `/proc` (similarly to `top`). In general, T_1 should be the time to enter the password, but we may initiate success detection earlier with little harm. For `su`, success results in the creation of a new shell process. T_2 should be the time to enter the password plus the time to start entering the password again if it's incorrect (terminal displays error, user reads it, user starts to type again). However, we do not remap too late, or the user may notice when she starts typing in the shell. For `su`, there is a delay in creating the shell process. If we detect the new shell (logon is successful) we map the key back immediately and measure the time expired between `su` detection (T_0) and logon detection. Otherwise we map back at T_2 but still must measure the time until new shell creation.

The results of our experiments are in Table 6. They confirm that the above algorithm performs as expected, and we observed no errors. The third column of the table gives the time between detection of the `su` client and successful logon (observed by detection of the bash shell). Since time durations are on the order of seconds, removal of accurate clocks will not prevent this channel.

Although errors are unlikely, there is the possibility of mapping the keyboard back before the user has entered the password. Most likely the last character would be affected; this character would be completely unknown to the adversary. Thus the adversary's guessing attack complexity would be increased in this case (i.e., by a factor of roughly 2.5). The guessing attack would still be tractable (less than a week).

Test	Remapped Key in Password?	Time until Logon
1	Yes	22.1 seconds
2	Yes	17.3 seconds
3	Yes	19.7 seconds
4	Yes	19.6 seconds
5	Yes	16.3 seconds
6	No	7.8 seconds
7	No	6.1 seconds
8	No	6.25 seconds
9	No	6.6 seconds
10	No	6.2 seconds

Table 6. Test Results Confirming Logon Detection Channel

5.2. Optimal Remapping Strategy

The adversary can select the number of keys to remap for each experiment. The optimal strategy is for the adversary to remap a small number of keys per experiment, given the constraint of limiting the number of remap-caused logon failures per unit time (e.g., suppose one remap caused

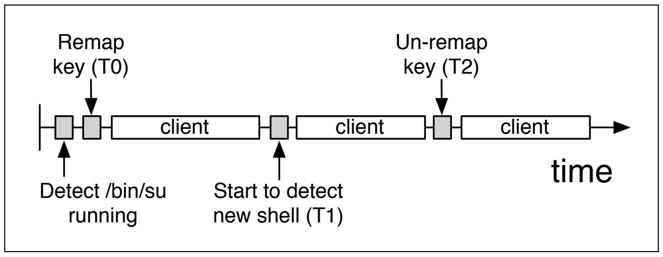

Figure 4. Logon Detection: Decision points are when to remap key (T_0), when to look for success (T_1), and when to un-remap key (T_2).

logon failure per week is permissible). Intuitively, remapping a larger set of keys, S, results in more remap-caused logon failures since S has a higher probability of intersecting the password. Then subsequent tests will need to narrow down within the set S to determine the actual characters within the password. On the other hand, if the keys in S were remapped one per test, then the majority of tests would not cause logon failures and can be executed without extra delays between tests. Also, this latter strategy avoids the overhead of the initial logon failure. In the appendix, we formalize this notion and demonstate an optimal strategy that remaps a single key per test, given suitable paramaters.

6. Guessing Attacks and Target Environments

For the timing experiments above, it is more efficient to reduce the possible password space to a size such that the password can easily be guessed, vs. attempting to completely determine the password via the timing experiments. The reason is that the experiments can only be run as often as the user performs a logon (at most several times per day). The main constraint is that the guessing program must be unprivileged.

To test the rate of password guessing, we wrote a guessing program, *pass*, that uses the `su` program as a password oracle. We used Expect to implement *pass*. We also wrote a Perl script which creates many parallel instances of Expect where each one sends different passwords to `su`. We tested the script to measure the password guessing rate, and this rate is used to find the average time to guess a password in the reduced keyspace.

The average guessing rate using this script is 90 guesses per second on the Linux 2.6 machine described above. By implementing the scripts in a lower level language such as C, we can obtain a significant performance improvement. For our estimates, we have assumed 100 guesses per second, and then reduced this number to 80 guesses per second based on the (simplified) assumption that our guessing program would consume approximately 80% of the CPU.

7. Countermeasures

A full exploration of countermeasures is beyond the scope of this paper. As mentioned above, the logon detection channel cannot be defeated by eliminating the hardware cache mechanism [10, 17]. Also, removing accurate clocks [3, 4] is unlikely to close this channel either.

One potential countermeasure is to use a trusted path mechanism (which is invoked by the user using a special key sequence). The keyboard can be remapped to a default configuration once the trusted path processing has been initiated. The original keyboard configuration can be restored upon exiting trusted path processing. During trusted path processing, only trusted processes should be allowed to run.

More generally, there is a need for methodologies that, instead of focusing on closing a particular channel, give generalized confidentiality assurance.

8. Discussion

Our Linux 2.6 timing framework may be of independent interest. It gives a task (thread) level timing capability (tasks are the scheduling unit in Linux 2.6). In other words, it allows us to accurately measure the CPU usage of other tasks on the system. Using the particular characteristics of X Window processing, we have been able to measure the CPU usage of a target X client.

We now discuss errors resulting from user keyboard entry mistakes. For logon detection, the most likely error is a mis-type of a key that is not remapped, causing it to accidentally be added to the adversary's password character set. The impact is roughly a doubling of the search space which is acceptable for one or two errors. For the other channels, the main impact would be if a remapped key is mistakenly entered in place of a non-remapped key. This event is very unlikely if only one or a small number of keys are remapped at a time. If it did occur, it would force additional tests, or the effects could be obviated by combining with the logon detection channel.

We did not test on a dual-core system. The logon detection channel should work as is on a dual-core system. The basic remap channel may require modifications such as running an additional process or additional threads. We leave this topic as future work.

For the logon detection channel, there is an HTTPS version (password authentication over a TLS/SSL channel). A local process can remap the keyboard while a network eavesdropper confederate can observe whether logon is successful on the HTTPS server. Potential issues include when to initially remap the keyboard and what type of follow-up attack is possible in order to disclose the password. We leave this topic as future work.

9. Conclusions

We have demonstrated new side channels on multiple versions of the Linux operating system, aimed at password disclosure. These channels limit the number of times a password can be used before disclosure. These channels require a locally running, unprivileged, process on the same host which the user enters keyboard input on. The mechanism for the channels is the X Windows keyboard remapping utility. One channel does not depend on the hardware cache.

The Linux 2.6 scheduler is easier to control than the Linux 2.4 scheduler. Our work reinforces the notion that side channels are of concern to both cryptographers and computer security designers. Future work includes approaches that are capable of demonstrating confidentiality in a general manner rather than simply closing specific channels.

References

[1] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson, TENEX, A Paged Time Sharing System for the PDP-10 *Communications of the ACM*, Vol. 15, pp. 135-143, March 1972.

[2] D.J. Bernstein. Cache-timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[3] Wei Hu. 1991, Reducing Timing Channels with Fuzzy Time. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1991, Oakland, CA.

[4] Wei Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1992, Oakland, CA.

[5] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO,* pp. 104-113, 1996.

[6] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM,* v.16 n.10, pp.613-615, Oct. 1973.

[7] B. W. Lampson. Hints for computer system design. *ACM Operating Systems Review*, 15(5):33-48, Oct. 1983.

[8] Adrian Nye. Xlib Programming Manual. Volume 1. O'Reilly, 1992.

[9] D. A. Osvik, A. Shamir and E. Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005.

[10] D. Page. Partitioned Cache Architecture as a Side-Channel Defense Mechanism. *Cryptology ePrint Archive,* Report 2005/280, 2005.

[11] Colin Percival. Cache missing for fun and profit 2005

[12] Werner Schindler. Optimized Timing Attacks against Public Key Cryptosystems. Statistics and Decisions, 20:191-210, 2002.

[13] Gaurav Shah, Andres Molina, and Matt Blaze. Keyboards and Covert Channels. In *Proceedings of the* 15^{th} *USENIX Security Symposium*. August 2006.

[14] Jonathan Trostle. Timing Attacks Against Trusted Path. In *IEEE Symposium on Security and Privacy*, pp. 125-134, May 1998.

[15] Jonathan Trostle. Modelling a Fuzzy Time System. In *Journal of Computer Security*, v.2, n.4, pp.291-310, 1993.

[16] Trusted Computer System Evaluation Criteria. United States Department of Defense. DoD Standard 5200.28-STD. December 1985.

[17] Zhenghong Wang and Ruby Lee. Covert and Side Channels due to Processor Architecture. In *22nd Annual Computer Security Applications Conference* December 11-15, 2006.

A. Optimal Strategy for Logon Detection Channel

Given a password alphabet of size A, a password with c characters (we assume the characters are distinct for simplicity), and let π be a remapping strategy for the adversary. Then we define $C(A, c, \pi)$ to be the cost (measured as number of logons needed) for obtaining the characters in the password given an alphabet of size A, a password with c distinct characters, and the strategy π. Also, $C(X, y) = min_{\pi} C(X, y, \pi)$. If $A = 96$, $c = 8$, and the delay for a remap caused logon failure is equivalent to the time it takes for 16 (non-remap failure) logons to be performed, then we can show that the optimal strategy is to remap one key per test. (If we decrease the delay time from 16, then we may obtain a strategy where it is sometimes beneficial to remap 2 keys per test.)

We obtain the equation:

$$\begin{aligned} & C(A, c, \pi_r) \\ = \quad & (1-\alpha)\left(\sum_i p_i(C(r, i|K) + C(A-r, c-i))\right) \\ + \quad & \alpha C(A-r, c) \end{aligned}$$

where π_r is the strategy that remaps r characters on the first test, and is optimal for succeeding tests, p_i = probability of i intersections given that a single intersection occurs, $C(X, y|K)$ is $C(X, y)$ conditioned on knowledge K from preceding tests, and $\alpha = (1 - c/A)(1 - c/(A-1)) \ldots (1 - c/(A - r - 1))$. Thus α is the probability that none of the remapped characters is in the password.

The idea behind the proof is that each remap test, using r keys, divides the set of password characters into two sets, one with r characters, and one with $A - r$ characters. Thus induction can be applied. We omit the details due to space limits.