Process Table Covert Channels: Exploitation and Countermeasures

Jean-Michel Cioranesco[1], Houda Ferradi[2], Rémi Géraud[2], and David Naccache[2]

[1] Rambus France, 54 Avenue Hoche, 75008 Paris, France
jcioranesco@rambus.com
[2] École normale supérieure, Équipe de cryptographie, 45 rue d'Ulm, F-75230 Paris CEDEX 05, France
given_name.family_name@ens.fr

Abstract. How to securely run untrusted software? A typical answer is to try to isolate the actual effects this software might have. Such counter-measures can take the form of memory segmentation, sandboxing or virtualisation. Besides controlling potential damage this software might do, such methods try to prevent programs from peering into other running programs' operation and memory.
As programs, no matter how many layers of indirection in place, are really being run, they consume resources. Should this resource usage be precisely monitored, malicious programs might be able to communicate in spite of software protections.
We demonstrate the existence of such a covert channel bypassing isolations techniques and IPC policies. This covert channel that works over all major consumer OSes (Windows, Linux, MacOS) and relies on exploitation of the process table. We measure the bandwidth of this channel and suggest countermeasures.

1 Introduction

A process table is a data structure in RAM holding information about the processes currently handled by an operating system. This information is generally considered harmless and visible to all processes and to all users, with only minor exceptions, on a vanilla system[3]. However, as pointed out by Qian et al. [13]:

> *"Even though OS statistics are aggregated and seemingly harmless, they can leak critical internal network/system state through unexpected interactions from the on-device malware and the off-path attacker"*

[3] Somes patches, such as `grsecurity` for Linux, may restrict the visibility of the process table. However, `grsec`'s default configuration doesn't affect our discussion.

Indeed, several papers [5, 13, 17] used data from the `procfs` on Linux systems to compromise network or software security. Namely, [17] could recover user's keystrokes based on registry information, and [5,13] accessed other programs' memory to mount a network attack. In all such cases, attacks relied on additional information about the target process (such as instruction pointers or register values) which were publicly available. Following these attacks the `procfs` default access right policy was changed in recent Linux versions.

This paper describes and analyzes a new covert channel exploiting the process table that can be used reliably and stealthily to bypass process isolation mechanisms, thereby allowing inter-process communication on all major operating systems. Malicious programs exploiting this strategy do not need any specific permissions from the underlying operating system. Contrary to earlier attacks, we do not assume any additional information (registry values etc.) to be available.

Prior Work. Whilst, to the best of the authors' knowledge, the problem of covert channel communication through process IDs (PIDs) was not formally addressed so far, the intuition that such channels exist must have been floating around, since most modern OSes currently randomize their process IDs. Interestingly, as we will later show, randomization makes our attacks *easier*.

Most known attacks on the process table exploited public information such as registry values [5,13,17]. Such information can be used directly or indirectly to recover sensitive data such as keys or keystrokes.

A long-standing bug of the BSD `procfs` became widely known in 1997 and relied on the possibility to *write* in the `procfs`, due to incorrect access right management. In that scenario, an unprivileged process A forks off a process B. Now A opens `/proc/pid-of-B/mem`, and B executes a setuid binary. Though B now has a different euid than A, A is still able to control B's memory via `/proc/pid-of-B/mem` descriptor. Therefore A can change B's flow of execution in an arbitrary way.

Besides these design flaws, many implementation mistakes led to a wealth of practical and powerful exploits against BSD's `procfs` in the early 2000's [2, 11, 12, 16].

There is also trace of an old Denial-of-Service remote attack dubbed the "process table attack" [1, 7]. According to [10, p. 93] this attack was developed by the MIT Lincoln Labs for DARPA to be used as part of intrusion detection systems. Their approach relies on the hypothesis that the only limit to how many TCP connections are handled is the number

of processes that the server can fork. This is by no means still the case on modern systems, rendering this attack completely inoperant. Note that instead of causing a DoS, the same approach could be used as a covert channel [3, p. 109].

2 Preliminaries

2.1 Covert Channels

Covert channels were introduced in [8], and subsequently analyzed in [4, 9, 14]. They are communication channels that enable communication between processes, which are not supposed to interact as per the computer's access control policy. We stress that the notion of covert channels is distinct from that of (legitimate) communication channels that are subjected to access controls. Covert channels are also distinct from side channels, which enable an attacker to gather information about an entity without this entity's collaboration.

Detecting covert channels is unfortunately generally hard, although general methodologies for doing so exist (e.g. [6]). Indeed, such channels may have devastating effects. Modern platforms implement a variety of security features meant to isolate processes and thus prevent programs from communicating, unless authorized by the security policy.

When modern counter-measures are not available, e.g. on mobile platforms, protecting against covert channels is very challenging. To further complicate things, many stake-holders take part in the development of mobile software and hardware (e.g. OEM handset manufacturers, telecommunication providers or carriers, application developers, and the device owner). For lack of better solutions, trusted execution environments (TEE) such as TrustZone emerged. These TEEs rely on hardware security resources present in the mobile platform which are not necessarily accessible to application developers and end-users.

2.2 Process IDs, Process Table, and Forking

Processes running on top of an OS are given a unique identifier called process ID, or PID. The PID enables the OS to monitor which programs are running, manage their permissions, perform maintenance tasks, and control inter-process communication.

Historically, PIDs were allocated in sequence: Starting at 0 and incremented until a system-specific maximum value, skipping over PIDs that belong to running programs. On some systems such as MPE/iX the lowest

available PID is used, in an effort to minimize the number of process information kernel pages in memory. Every process knows its own PID[4].

Complex applications also leverage process IDs. One typical example is forking: A process creates a copy of itself, which now runs alongside its parent. The PID of a parent process is known to the child process[5], while the child's PID — different from that of the parent — is returned to the parent. The parent may, for example, wait for the child to terminate[6], or terminate the child process. Between the moment a child process dies, and its parent reaps its return value, the child process is in a special *zombie* state[7].

Fork is the primary method of process creation on Unix-like operating systems, and is available since the very first version of Unix [15]. For DOS/Windows systems lacking fork support, the almost equivalent `spawn` functionality was supplied as a replacement for the fork-exec combination.

On Unix-like systems, information about all currently running processes (including memory layout, current execution point, open file descriptors) is stored in a structure called the process table. Whenever a process forks to create a child, the entire process table entry is duplicated, which includes the open file entries and the their file pointers — this is in particular how two processes, the parent and the child, can share an open file.

By default on Unix-like systems, the complete process table is public and accessible as a file through the `procfs`. Alternatively, non-root users can execute the `ps -efl` command to access the detailed table. On Microsoft Windows platforms (XP and above) the list is accessible through the `EnumProcesses` API[8].

For the sake of simplicity we used Python 3.4 with the `psutil` library to abstract these implementation details away. We thus have a wrapper function `ps` that works on all major platforms and provides us with process information.

---

[4] For instance using the `getpid()` system call on Unix-like OSes, or `GetCurrentProcessId()` on Windows platforms

[5] For instance using a `getppid()` system call on Unixes.

[6] For instance using the `waitpid()` function on Unixes.

[7] See the Unix System V Manual entry: http://www-cdf.fnal.gov/offline/UNIX_Concepts/concepts.zombies.txt.

[8] See https://msdn.microsoft.com/en-us/library/windows/desktop/ms682623(v=vs.85).aspx.

3 Overview of the attack

3.1 Assumptions

We consider two programs A and B, and for the sake of clarity consider that A wants to send information to B. We assume that the operating system can freely implement any process isolation technique of its choosing, while allowing the following minimalistic assumptions:

1. A can fork[9];
2. B can see A and its forks in the process table.

Even though there are restrictions on the number of forks that a process can launch, this limit is usually larger than one. In this work we only require the ability to launch *one fork* at a time. Forking at least once is nearly always possible. The second assumption is not unreasonable, as most systems expose all processes, including those launched by other (potentially priviledged) users and the kernel. Furthermore, tools such as `unhide`[10] try to detect hidden processes by comparing the outputs of different programs and looking for inconsistencies. Similar techniques could be implemented by B even if the OS tries to restrict process visibility.

The idea here is to exploit the fact that PIDs are public to construct a covert channel.

3.2 Naive approach

First, assume that no processes other than A and B are being run. When A forks, the sum of all visible PIDs increases. When that forked process dies, the PIDs' sum decreases. B queries the process table repeatedly and monitors the differences between successive sums – which are interpreted as 0s or 1s.

Now what happens to this approach when we remove the assumption that no other processes are running? New processes are launched by the OS and by users. Old processes die. This may cause process table modifications at any time.

Consequently, the aforementioned naive approach doesn't work anymore.

[9] Or, equivalently, A can launch at least a process and later kill it.
[10] See http://www.unhide-forensics.info/.

3.3 Handling noise

It might happen that when new processes are created, their PIDs are predictable – oftentimes the smallest available one. To some extent, this information could be used to deal with noise. However, such an approach would fail if processes are removed from the process table (and new ones start reusing the freed spots), and therefore wouldn't be reliable over time. Here we make no assumptions on the PIDs' distribution, and assume for the sake of simplicity that PIDs are distributed uniformly.

Let p be a prime number. If x and y are distributed uniformly modulo p, then $x + y$ is also distributed uniformly modulo p. We make use of this fact in the following way: when a process is created or deleted, the sum of the PIDs modulo p changes to a value S which is, by the previous remark, chosen uniformly at random modulo p.

Let T be a target value, consider the following algorithm.

1. Let $f \leftarrow 0$.
2. If $S = T$ go to 1.
3. If $S \neq T$ and $f = 0$, the main process A creates a fork[11] A'.
4. If $S \neq T$ and $f = 1$, the process A' kills itself.

Note that, assuming that there is no noise, this succeeds in setting $S = T$ in expected p iterations. In other terms, if there is no external process creation or deletion during p iterations, S is set to the target value T.

The strategy consists in running this algorithm continuously. Whenever there is a change in the process table, forks are created or deleted until the desired target sum is reached.

Note that, in the absence of noise, and if the OS attributes PIDs *deterministically*, then this algorithm may fail, as it could be stuck oscillating between two incorrect S values.

3.4 Channel capacity

Assume that external process creation or deletion happens on average every Δ time units (one could consider a Poisson distribution for instance). If it takes a time t_S for A to query S and t_f to fork (or kill a fork), then it takes an expected time $p(t_S + t_f)$ to reach the target value. Therefore this algorithm sends $\lfloor \Delta / p(t_S + t_f) \rfloor / \Delta$ elements of $\mathbb{Z}_p$ per elementary time unit. Hence, channel capacity is:

$$C(p) = \frac{\lfloor \Delta / p(t_S + t_f) \rfloor \log_2 p}{\Delta} \text{ bit/s}$$

[11] It is assumed that the forked process knows that $f = 0$. This value could be sent as command line argument for instance.

Note that Δ should be large enough, namely $\Delta > p(t_S + t_f)$, for the channel to allow sending data at all. $C(p)$ is maximal for $p = 2$, which incidentally makes implementation easier.

If one wishes to send data faster, error-correcting codes (such as LPDC) may be used to thwart the effect of noise and make communication reliable at higher rates.

4 Experimental setup

A proof-of-concept was implementated in Python 3.4, using the `psutil` library. Code was tested on a Linux server (Debian Jessie) and Microsoft Windows 7 and 8, for both 32-bit and 64-bit architectures, with similar results. All test machines were active web servers running Apache or Microsoft IIS in their latest versions as we are writing these lines.

The proof-of-concept consists in two programs, a *sender* and a *receiver*, that may be granted different permissions and be launched by different users. The implementation follows straightforwardly Section 3. The test consisted in sending a given sequence of bits from the sender to the receiver. The observed sequence on a busy server is illustrated on Figure 1, where the target sequence was "`010101...`".

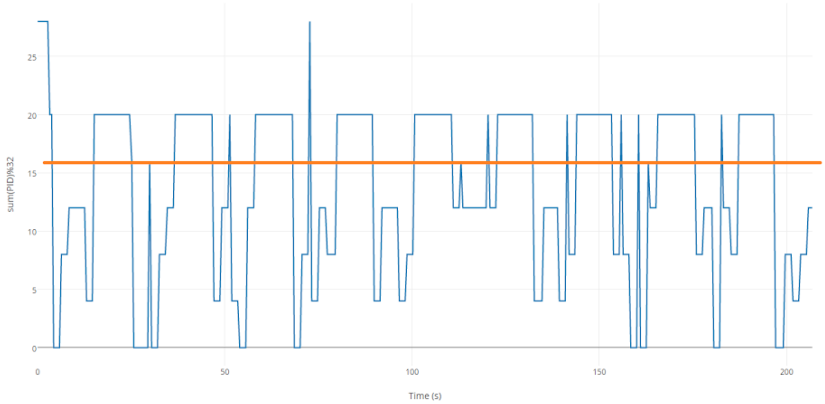

Fig. 1. Use of the PID covert channel on Windows 7 demonstrating how the message `010101...` can be sent.

5 Countermeasures

A number of solutions can be implemented at various levels to counter the attacks described in this paper. A prerequisite is the precise definition of the attack model. If we assume that the receiver and the sender use a *known* transmission process whose parameters are potentially known (for instance a key k shared between the sender and the receiver) then we can imagine *ad hoc* countermeasures targeting the specific transmission process and/or k. If, on the other hand the communication process is unknown then a number of generic countermeasures can be devised to try and prevent unauthorized transmission in general.

Because an important number of information leakage methods can be imagined and designed, this section will only deal with generic countermeasures. Note that we *do not* claim that any of the generic methods below will have a guaranteed effect on *all* PID-based covert channels.

5.1 Restricting process visibility

A most natural approach is simply to make PID information invisible to processes. At first glance, restricting (even partially) process visibility solves the issue, as two mutually invisible processes cannot communicate as described in this paper. To some extent, this is the kind of strategy employed by security patches such as `grsec` for Linux.

However, such a policy has limits. Indeed, there are processes that *need* to communicate and IPC was precisely designed to enable that. The goal is not to prevent any process from communicating with any other process, but to allow so *if and only if* such communication is permitted by the OS's security policy. It is sensible to try and hide kernel-related and other sensitive processes from untrusted programs, however it is not a good idea to isolate all processes from one another.

This policy restriction has the disadvantage of grandly reducing system functionally. Furthermore this counter-measure is vulnerable to transitive attacks, whereby a process acts as a relay between two mutually invisible programs. In some instances, processes may bypass the process table altogether, for instance by attempting to directly contact random PIDs. Depending on the answer, it is possible to guess that a process is running with that PID, even though it cannot be seen in the process table. Alternatively, it is possible to run legitimate commands (e.g. `lsof`) which have a different view of the process table. By using such commands, processes can gather information otherwise denied to them.

5.2 Zombies, timing and decoys

The most evident idea is similar to the adding of random delays in timing attacks or to the adding of random power consumption to prevent power attacks. Here the operating system can randomly launch and stop processes to prevent B from properly decoding the information coming from A. Note that the OS does not need to launch real processes, only *zombies* i.e. PIDs present in the table that do not correspond to actual processes. Alternatively, the OS may simply add random PIDs (decoys) when replying to a query about the process table.

To be efficient, this countermeasure needs to generate a sequence of process starts and interrupts in a way that effectively prevents information decoding by B. To illustrate our purpose, assume that B's processes are very rarely killed, and that the OS launches and kills PIDs at a very high pace. After a sufficiently long observation time, B may infer the processes belonging to A. As this is done A may start communicating information to B by progressively killing the processes it controls. This illustrates the need to have the OS generate and kill PIDs in a way indistinguishable from A. Because we do not know *a priori* the communication conventions used for this covert channel, this countermeasure can only rely on empirical estimates of normal program behaviour.

If the time between launched processes is used to send information, the OS can randomly delay the removal of PIDs from the list to prevent communication based on PID presence time.

A number of generic approaches, inspired by fraud detection, can also be imagined. The idea here consists in limiting system performance to reduce the attacker's degree of freedom. For instance, limiting the number of offspring processes launchable per unit of time by a process is also likely to have an effect on the attack as it would naturally reduce information rate. Note that this restriction should only apply to processes whose launcher requires a PID to appear publicly. Fraud-detection countermeasures consist in monitoring the frequency at which the PID-list is read by each process and detect processes whose behaviour may betray the reading of a covert channel.

5.3 Virtual PIDs

By modifying the way in which the OS manages PIDs, other countermeasures can be imagined. A possible way to implement such a protection consists in having a private PID list per process. Here, process U sees the PID of process V as $f(s, U, V)$ where s is an OS secret known to processes

U and V. This allows U to solicit V without sharing PID information visible by both U and V. This may reduce the available information transmission channels to *global information* such as the PID-list's cardinality (number of processes), the time separating the appearing of new PIDs in the list etc.

A variant works as follows:

- *Each time* a process U queries the process table, U is given a random list of PIDs.
- When U requests an IPC with some process V, then upon the OS's IPC approval, the PIDs of V and U as seen by each other do not change anymore.

This still provides IPC functionality while preventing process table abuse.

5.4 Formal proofs

To tackle the problem in general, a formal model should be defined, so that one can attempt to come up with *proofs* of isolation. For instance, a process may be modelled as the data of a process birth time, a process death time and a value assigned by the OS. The birth and death times are controlled by the sender and the analyst's goal is to determine the channel capacity in the presence of generic countermeasures.

To the best of our knowledge, such models have not been developed so far.

References

1. DARPA: DARPA Intrusion Detection Evaluation (2000), https://www.ll.mit.edu/ideval/docs/attackDB.html
2. Etelavuori, E.: Exploiting Kernel Buffer Overflows FreeBSD Style: Defeating security levels and breaking out of jail(2), http://ftp.ntua.gr/mirror/technotronic/newfiles/freebsd-procfs.txt
3. Gligor, V.D.: A guide to understanding covert channel analysis of trusted systems. The Center (1994)
4. Huskamp, J.C.: Covert communication channels in timesharing systems. Ph.D. thesis, University of California (1978), Technical Report UCB-CS-78- 02
5. Jana, S., Shmatikov, V.: Memento: Learning secrets from process footprints. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 143–157. IEEE (2012)
6. Kemmerer, R.A.: Shared resource matrix methodology: An approach to identifying storage and timing channels. ACM Transactions on Computer Systems (TOCS) 1(3), 256–277 (1983)
7. Kendall, K.R.: A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems (1999), MIT Master's Thesis. Available at https://www.ll.mit.edu/ideval/files/kkendall_thesis.pdf

8. Lampson, B.W.: A note on the confinement problem. Communications of the ACM 16(10), 613–615 (1973)
9. Lipner, S.B.: A comment on the confinement problem. In: ACM SIGOPS Operating Systems Review. vol. 9, pp. 192–196. ACM (1975)
10. Marchette, D.J.: Computer intrusion detection and network monitoring: a statistical viewpoint. Springer Science & Business Media (2001)
11. Nash, A., Newsham, T.: A bug in the procfs filesystem code allows people to modify the (priviliged) init process and reduce the system securelevel. (1997), http://insecure.org/sploits/BSD.procfs.securelevel.subversion.html
12. memory disclosure in procfs, K., linprocfs: (2004), https://www.freebsd.org/security/advisories/FreeBSD-SA-04:17.procfs.asc
13. Qian, Z., Mao, Z.M., Xie, Y.: Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 593–604. ACM (2012)
14. Schaefer, M., Gold, B., Linde, R., Scheid, J.: Program confinement in KVM/370. In: Proceedings of the 1977 annual conference. pp. 404–410. ACM (1977)
15. Thompson, K., Ritchie, D.: Sys fork (ii)
16. Thorpe, J., Hannum, C., Jones, C., van der Linden, F.: NetBSD Security Advisory 2000-001: procfs security hole (2000), ftp://ftp.netbsd.org/pub/NetBSD/misc/security/advisories/NetBSD-SA2000-001.txt.asc
17. Zhang, K., Wang, X.: Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In: Proceedings of the 18th Conference on USENIX Security Symposium. pp. 17–32. SSYM'09, USENIX Association, Berkeley, CA, USA (2009)