Towards secure user-space provenance capture

Nikilesh Balakrishnan Thomas Bytheway Lucian Carata Ripduman Sohan Andy Hopper
University of Cambridge
firstname.lastname@cl.cam.ac.uk

Abstract

System and library call interception performed entirely in user-space is a viable technique for provenance capture. The primary advantages of such an approach are that it is lightweight, has a low barrier to adoption and does not require root privileges to install and configure. However, since both the user's application and the provenance capture mechanism execute at the same privilege level and as part of the same address there is ample opportunity for an untrustworthy user or application to either circumvent or falsify provenance during capture.

We describe a security threat model for such provenance capture mechanisms, discuss various attack vectors to circumvent or falsify provenance collection and finally argue that hardening against such attacks is possible if the application is sandboxed using contemporary techniques in the area of user-space software-based fault isolation.

Categories and Subject Descriptors 500 [*Information systems*]: Data provenance

General Terms provenance, security, sandbox

Keywords provenance, security, sandbox

1. Introduction

Provenance capture by observing applications as they execute and logging events of interest in a structured, queryable format can be implemented using a variety of approaches. Most systems implementers have preferred the approach of capturing provenance in the kernel by intercepting system calls [6–9]. This approach has several useful properties for a provenance system: (i) The capture is performed at a layer having a higher privilege level than that of the application, ensuring provenance collection is secure and non-circumventable (assuming that the kernel cannot be compromised). (ii) Provenance capture is always-on and system-wide. (iii) The runtime overhead imposed by such a mechanism is reasonably low. However, a major obstacle to this approach is that these systems are not easily deployable since the capture mechanisms themselves rely on the installation of customised kernels and new kernel features which may not be available widely, thereby resulting in a lack of adoption in practice.

In previous work [1] we have argued that the problem of lack of adoption of provenance systems can be mitigated by designing systems that work entirely in user-space wherein the provenance capture is done at the same privilege level and as part of the same address space as that of the application. User-space provenance systems have a number of advantages that make it attractive for an implementer:

- The system becomes easier to deploy requiring no root privileges to install and configure.
- The system can work on a variety of kernel versions or platforms since it does not rely on specific kernel features or modifications to kernel internals.
- The runtime overheads imposed can be comparable to that of kernel based mechanisms.
- The system can be developed in a much richer ecosystem, allowing for new features to be implemented quickly.

However, in order for provenance to be useful it must be trustable. The implicit assumption in the user-space model of provenance capture is that the application or user can be trusted. This however may not be the case in practice since we cannot guarantee that users and applications are trustworthy/non-malicious. A user may wish to hide his activities from the system by circumventing it completely or forge results of his experiment by falsifying provenance data at the point of collection. These capabilities defeat the purpose of provenance capture in the first place and render the system ineffective. This leads us to the research question, *"Is secure user-space provenance capture possible in principle? What are the design requirements and limitations for such a system?"*.

There is a plethora of research in computer systems security that looks at a variety of attack scenarios that can be carried out by a malicious user and strategies to harden against such attacks. Most of these scenarios assume that the application code is trusted and that an attack happens by injecting malicious code into the application's address space. The application is then tricked into executing this malicious code to gain control of the system. However, for secure user-space provenance capture the problem is more challenging since the user or application itself cannot be trusted. This increases the attack surface and conventional techniques to harden against security exploits may not be sufficient in this context.

In this paper we discuss various techniques for capturing provenance in user-space and their characteristics. We then focus on a specific capture technique relevant to our system based on library interposition and explore various attacks that a malicious user or application could perform to circumvent the system or falsify provenance data. Finally, we review application sandboxing techniques based on software fault isolation (SFI) that user-space provenance systems (based on system/library call interception) could employ to harden against such attacks. We believe that the application of sandboxing in the context of secure user-space provenance capture is a novel use-case and can open up a new direction of research for the provenance community.

2. User-space provenance capture

Fine-grained tracing of applications has been an active area of research both in academia and industry. Over the years a number of techniques and primitives have been introduced to enable tracing of user-level applications. Several tools have been developed on top of

TaPP 2016, June 8–9, 2016, Washington D.C., USA.

these techniques to enable developers and users to debug and profile their applications. From a systems perspective these same techniques can be leveraged to collect provenance meta-data (typically system call and library API call data).

However, several of these user-space tracing techniques were not designed for the requirement of an always-on, system-wide, complete provenance collection system nor were they designed for the secure collection of provenance meta-data. Therefore as implementers it is important to understand the characteristics of each of these techniques. In the rest of the section we will briefly describe several techniques available for user-space provenance capture and discuss the advantages and disadvantages of each.

FUSE: is an interface that allows non-privileged users to create a custom user-space file system. FUSE consists of two main components, a kernel module that acts as a bridge to kernel interfaces and a user level library libfuse that a user-space provenance capture agent can link with to implement wrappers for various I/O activities that an application can perform. Installation of FUSE requires the user to have root permissions. Also from a performance standpoint, FUSE can incur a high overhead for certain workloads [5]. However, the main drawback of using FUSE for provenance capture is that only I/O operations can be intercepted, other process level activities and information pertaining to the process environment are not visible to the system.

PTrace: is a system call that is supported by most Unix-like operating systems. PTrace provides a mechanism by which one process, a tracer may observe and control the execution of another process. With this technique whenever a specific event of interest (e.g. system calls) occurs during the target programs execution, the kernel stops the execution of the target process and transfers control to the tracer. The tracer can then examine the target's internal state such as registers or memory and collect required information. This can be a powerful technique for provenance capture as it provides complete visibility into a program's execution. An additional benefit of using ptrace is that it does not require root privileges to be enabled unless the target process being traced is running as root. However, a major drawback with ptrace is the high runtime overheads (˜30%) it can impose on applications due to the two context switches, from target to tracer and back [2]. This makes PTrace impractical to use from the point of view of implementing an always-on, system-wide provenance system.

Kernel-assisted user-space tracing: Over the years Linux contributors have recognised the need for a flexible tracing infrastructure for both the kernel and userland applications. As a result of this, support for a number of mechanisms/tools such as SystemTap, DTrace, Uprobes etc. have arisen in newer versions of the Linux kernel. With these mechanisms a probe in the form of a kernel module is typically loaded to monitor specific events e.g. system calls that applications invoke. A benefit of this approach is that provenance collection happens within the kernel which executes at a higher privilege level and and can therefore be deemed secure. A user will however require root privileges to enable and install these probes. The major drawback of these mechanisms however is that support for such kernel-assisted tracing functionality may not be widely available on older kernel versions.

LD_PRELOAD: is part of the linker/loader's mechanism for extending the environment of a process at load time and is a lightweight method for interposing library function calls. The LD_PRELOAD environment variable specifies a list of shared library paths and the runtime loader checks for the presence of this variable and loads the list of libraries into the address space of the application it is trying to bootstrap. During runtime the linker/loader resolves symbols by searching for the symbol within the preloaded list of libraries first, thereby providing an easy method to hook API functions for a given target library. This is a viable method for provenance capture since file and process related function calls made to the standard C library can be interposed from within the application's address space. The runtime overheads incurred by this mechanism is relatively lower (˜10%) when compared to ptrace. Also to enable this mechanism a user need not require root permissions. This makes the deployment of such systems much easier. However, a major drawback for systems that use this mechanism for provenance capture is that a malicious user or program can perform a variety of exploits to circumvent the system completely or falsify provenance data at point of capture.

Binary rewriting: is the process of transforming executables by adding or modifying individual instructions to implement a specific requirement while still maintaining the binary's original functionality. This can be performed statically by disassembling the binary and patching required instructions or dynamically by translating basic blocks using a just-in-time compilation approach. Provenance systems could use this mechanism to rewrite specific instructions, for example the syscall instruction and redirect the system call to a trampoline where required meta-data may be collected. Manually rewriting the binary by deciphering instruction opcodes may be a difficult exercise. However, there are a number of tools/libraries available today that ease this development effort by exposing a simpler interface [10, 11]. A major benefit of using this technique for provenance capture is that the system can be implemented entirely in user-space without requiring root privileges. However, since the provenance module executes as part of the same address space and privilege level as that of the application, a malicious application could tamper with the provenance module's internal state or falsify the provenance data captured.
Our focus for the next section will be to understand the security vulnerabilities of user-space provenance capture especially for those mechanisms where capture occurs within the application's address space. Therefore the attacks we describe are mainly applicable to systems that use LD_PRELOAD. However, a few attacks (DoS and TOCTTOU) are applicable to binary rewriting as well.

3. Vulnerabilities of library level interposition

3.1 Threat Model & Assumptions

We consider a user that has logged into a machine using his account. Provenance for all user activities for the session is captured using library level interposition. The provenance is stored in a backend database under a different userid on the same machine. For such user-space provenance system there are two types of threats that we must consider:

- Circumvention: The user may attempt to hide his activities by completely circumventing the provenance system. For example, the user could attempt to obtain a new uninterposed shell session by invoking a system call instruction that executes the /bin/bash program.
- Falsification: (i) The user could try to trick the system into collecting false provenance information. For example, a user may modify the file path passed to the `open` function call by using a custom library in-between the provenance capture and standard C library. (ii) The user may also attempt to forge provenance records at point of collection. Since an application has access to its entire address space, including the data sections of the provenance collection library, it could modify provenance records directly in memory.

We define the trusted computing base (TCB) to be the kernel, the provenance collection library, the provenance storage backend, the runtime linker/loader and the standard C library available as part of the system. We consider all user level application binaries and user supplied libraries to be untrusted. We assume that the user cannot tamper with provenance data once it leaves the application's address space therefore any in-transit or stored provenance data is considered safe from any exploits. We can justify this based on our assumption that the kernel is trusted and therefore any IPC mediated by the kernel is safe from exploits.

3.2 Types of attacks

Direct calls to standard C library: When an application invokes a library function using its exported symbol for e.g. `printf` the compiler generates code to call a trampoline entry in the the PLT (Procedure Link Table), e.g. `call printf@plt`. The code in the the PLT invokes the linker and resolves the address of the symbol. This level of indirection is required because shared libraries are typically compiled as PIC (Position-independent code) and the runtime address of symbols in the shared library cannot be computed at compile time. This indirection also enables the preloading of shared libraries passed via the LD_PRELOAD environment variable or the /etc/ld.so.preload configuration file. When the preload option is enabled, the runtime linker first searches the preload library list to resolve the symbol, allowing us to interpose any exported symbol used in shared libraries.

An application can however skip this symbol resolution mechanism and call or jump directly to the address of a function in a shared library for e.g. the `printf` implementation in the standard C library, effectively circumventing the interposition layer that collects provenance. This attack can be implemented in two ways.

The first technique uses the interface exposed by the dynamic linker. An application could use the linker's `dlsym` function to lookup the address of `printf` within the standard C library and call this address using a function pointer. A simple fix to harden against this attack is to override the interface exposed by the dynamic linker.

The other method of obtaining the address of the actual `printf` function is to read the /proc/pid/maps entry for the process and compute the address where the code segment of the standard C library is loaded in memory. The application can then use the offset of the `printf` function from the binary to jump directly into a valid address within the function block after setting up the the stack. To harden against this attack jump instructions to addresses within the TCB should be disallowed.

Man-in-the-middle: The provenance collection mechanism relies on the linker's preload feature to interpose standard C library function calls. A malicious user looking to trick the provenance system into collecting false provenance data could employ the same interposition technique and implement a custom library that hooks library calls made by the application. A standard linker/loader implementation will load any valid shared object supplied to it using the preload mechanism as it assumes all user supplied libraries to be trusted.

To understand how this attack may be implemented let's look at the following scenario. A scientist is about to execute a program that reads a file `results.dat` as part of an experiment. However, the scientist would ideally like the program to read data from a `mod_results.dat` file which contains fudged data. To prove that his experiment was performed with genuine data he also needs to trick the system into capturing provenance for `results.dat` instead of `mod_results.dat`. To carry out this attack the scientist implements a custom library that has overridden the `open` function call. The custom library is then added to the LD_PRELOAD environment variable after the provenance collection library. When the application calls the `open` function with `results.dat` as argument, control is first transferred to the provenance capture library, which then forwards the request to the custom library. Within the custom library the argument to `open` is modified to use `mod_results.dat`. The provenance capture library is however unaware of this change and captures meta-data showing that the experiment was performed with `results.dat`. To harden against this attack a secure policy based linker/loader implementation is required to ensure that only a predefined set of trusted libraries are loaded.

Controlling the Instruction Pointer: The instruction pointer (IP) is a register that indicates the address of the instruction currently being executed by the processor. The IP is implicitly modified when a particular instruction is executed and cannot be manually set for example using a `mov` instruction. However if an application is being traced using the `ptrace` system call, the tracer program can manually set the IP to any valid address in the instruction sequence using the set registers feature. This immediately presents a security threat for the provenance system since the application could completely circumvent provenance capture by stepping over the instruction sequences that record and store provenance.

To harden against this attack the provenance system could either intercept the `ptrace` system call and disallow it or IP register manipulation in user-space could be completely disabled with processor support.

Direct system call invocation: The standard C library provides a convenient interface for application developers to interact with the kernel since it abstracts away the architecture and platform specific complexities of invoking system calls. However, there is nothing preventing a user-space application from directly invoking the system call instruction. This presents a problem for library interposition since direct system calls cannot be intercepted. A malicious application can use this capability to circumvent the system. For example, a user could implement a program that calls the architecture specific syscall instruction to invoke the `exec` system call and pass the bash path as argument. This will result in a new uninterposed shell session.

In order to harden against such attacks system calls must only be allowed from within the TCB, i.e the standard C library. A naive way to implement this in the kernel is for the syscall interface to check the source address from where the system call is being invoked and allow it if the source address is from a trusted part of memory. However, since the application can transfer control to any code segment loaded in memory, a `jmp` to a system call instruction sequence within the TCB will be sufficient to circumvent the system. This naturally leads to a model where the application executes within a sandbox and control integrity is enforced.

Denial of service (DoS): With user-space provenance capture, all modules (shared objects and the application binary) loaded in memory share access to a common pool of resources available for the process. This includes file descriptors, heap memory, environment variables etc. An application intending to disable provenance collection could launch a DoS attack either by completely exhausting the resources used by the provenance module or by tampering with the provenance module's internal state. For example, a malicious application may close a file descriptor used by the provenance module to communicate provenance data to a backend for storage by directly invoking the `close` system call. Similarly, a malicious application having access to the data section of the provenance module could modify the module's internal state directly.

This attack is possible because the application has unrestricted access to its address space and the capability to make system calls. To harden against this attack the provenance module's code and data can be isolated from the application. Also, the application should not be allowed to make system calls directly.

Time of check to time of use (TOCTTOU): The provenance capture module stores meta-data in memory before sending it to a storage backend. This immediately poses a direct threat for the integrity of provenance records because an application that shares the address space with the provenance module can tamper with this in-memory data. Techniques such as address space layout randomisation (ASLR) can make finding the exact location of data in memory difficult. However, an application that has access to its entire process memory map can compute the addresses of all code and data segments in memory and use this information to falsify provenance records.

To harden against such attacks the data section of the provenance collector should not be accessible from the application's binary. This can be implemented by sandboxing the application and restricting code and data access to only a part of the address space. Also, to ensure that buffer overflows occurring in the TCB do not overwrite provenance data, the memory pages where provenance records reside can be protected by guard pages that have read-only permissions. Writing to these guard pages will cause the application to segfault.

4. Secure user-space provenance capture

User-space provenance capture suffers from several security vulnerabilities as seen in the previous section. The attacks described previously arise as a result of (i) the application having unrestricted access to its entire address space, including all code and data and (ii) the capability of the application to execute any instruction that it has privileges for in user-space. Security mechanisms such as ASLR and policy based system call authorisation do no mitigate against these problems because they are designed to harden against attacks where malicious code is injected into the application to gain system access, they do not assume that the application itself is malicious.

To mitigate against such attacks we argue that user-space provenance systems should employ contemporary techniques in software-based fault isolation and restrict the execution of user application within a sandboxed environment. The sandboxing mechanism should enforce three key constraints on the application:

- Control flow integrity: All direct and indirect branches in the application code should reach a safe instruction within the sandbox. The application should not be allowed to branch directly to a region of code in the TCB. All control transfers to the TCB should happen via a validation layer.
- Data Integrity: No loads or stores should be permitted outside of the sandbox. Once the binary is loaded into memory it should not be writable.
- No unsafe instructions: The application should not be allowed to invoke any system call instructions directly.

4.1 Application sandboxing

Google's native client was one of the initial efforts to sandbox the execution of native x86 code inside web browsers [4]. The Native client technique for sandboxing ensures that direct and indirect branches in the application stay within bounds of the sandbox and land on a safe instruction. Similarly, for data access, load and store instructions are constrained to remain within the sandbox. This is implemented with the help of a custom compiler to add instrumentation to the application binary and a runtime validation process.

The techniques implemented by Native Client are an effective method for sandboxing applications, however, a major barrier to adopting this approach for desktop applications is the requirement for applications to be recompiled. More recent efforts [3] use dynamic binary translation where each basic block in the code is checked for the sandbox constraints at runtime and translated. The inherent advantage with this approach is that the translator processes one instruction at a time and can compute destination address of indirect branches and memory accesses easily at runtime, requiring no support from the compiler or the hardware. Also, attacks that manipulate the return address on the stack to transfer control can be thwarted by implementing a shadow stack and verifying the validity of the return address against the shadow stack before executing the `ret` instruction. However, dynamic binary translation can impose a high overhead on application runtime, especially if every load and store operation needs to be checked. To mitigate against this hardware support for sandboxing applications may be required.

4.2 A model for secure user-space provenance capture

Our model for the secure capture of provenance in user-space extends the the sandbox design for the safe execution of applications using software-based fault isolation based on dynamic binary translation [3]. In our model the user supplied application binary and shared libraries are considered untrusted and therefore executed within the sandbox. A secure policy based loader implementation ensures that the sandbox code is initialised first followed by the provenance collection library before bootstrapping the application binary and other shared libraries. The secure loader also ensures that only a predefined set of trusted shared libraries are allowed for preloading (using LD_PRELOAD). Finally, the secure loader also marks the code, PLT, GOT and other sections involved in dynamic linking for the untrusted code base as non-writable to avoid exploits.

The dynamic binary translator translates and validates every instruction before execution. Target addresses of each branch instruction (direct and indirect) are checked so that it lands on a safe instruction within the sandbox. Similarly, addresses used in loads and stores are computed and kept within bounds of the sandbox. Finally, instruction sequences that represent system calls are disallowed from inside the sandbox. Only modules in the trusted computing base are allowed to invoke the system call instruction. Any violation of these checks cause the program to terminate with a security exception. These runtime checks ensure that the constraints of the sandbox described previously are enforced.

However, under valid circumstances control transfers from the untrusted sandboxed modules to trusted modules are allowed, only via the `call` instruction. For example if the application wants to invoke an exported standard C library function. In such instances the control is first transferred to a trampoline function in the trusted runtime where the instruction is validated, the target address is checked to see if it is reachable and the symbol resolution via the GOT/PLT mechanism is allowed to proceed. Function pointer based calls to the standard C library are rewritten and redirected to relevant symbols in the provenance capture library. This ensures that the application cannot bypass the provenance module and circumvent the system.

Limitations of our model

- Applications may use pointers to access parts of global memory in other shared objects. For example, the `environ` pointer points to a list of environment variable data maintained within the standard C library (Glibc). If the memory pointed to is within the TCB the sandbox will prevent this access. To allow such accesses we will need to in effect have a model where the environment variables and data specific to the sandboxed application lives inside the sandboxed memory.
- The heap memory area for a process is common across all threads and modules in the process. Heap memory allocated by the modules in the trusted computing base should not be be accessible to the untrusted modules. This will require maintaining two heap regions, requiring a custom memory allocator.
- The requirement for a custom secure loader is a change to the user's system and will require root privileges to install.

5. Conclusion

In this paper we discussed the vulnerabilities of user-space provenance capture using library level interposition. We showed that these vulnerabilities arise as a result of a lack of separation between the provenance capture mechanism and the application. Finally, we argued that to harden against these vulnerabilities and enable secure capture of provenance, implementers could employ contemporary techniques in application sandboxing techniques.

References

[1] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2013. OPUS: a lightweight system for observational provenance in user space. In Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP '13). USENIX Association, Berkeley, CA, USA, Article 8, 4 pages.

[2] Philip J. Guo and Dawson Engler. 2011. CDE: using system call interposition to automatically create portable software packages. In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11). USENIX Association, Berkeley, CA, USA, 21-21.

[3] Mathias Payer and Thomas R. Gross. 2011. Fine-grained user-space security through virtualization. In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11). ACM, New York, NY, USA, 157-168. DOI=http://dx.doi.org/10.1145/1952682.1952703.

[4] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: a sandbox for portable, untrusted x86 native code. Commun. ACM 53, 1 (January 2010), 91-99. DOI=http://dx.doi.org/10.1145/1629175.1629203

[5] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. 2009. Story book: an efficient extensible provenance framework. In First workshop on on Theory and practice of provenance (TAPP'09). USENIX Association, Berkeley, CA, USA, , Article 11 , 10 pages.

[6] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware storage systems. In Proceedings of the annual conference on USENIX '06 Annual Technical Conference (ATEC '06). USENIX Association, Berkeley, CA, USA, 4-4.

[7] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In Proceedings of the 13th International Middleware Conference (Middleware '12), Priya Narasimhan and Peter Triantafillou (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 101-120.

[8] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: collecting high-fidelity whole-system provenance. In Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12). ACM, New York, NY, USA, 259-268. DOI=http://dx.doi.org/10.1145/2420950.2420989

[9] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the Linux kernel. In Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15), Jaeyeon Jung (Ed.). USENIX Association, Berkeley, CA, USA, 319-334.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40, 6 (June 2005), 190-200. DOI=http://dx.doi.org/10.1145/1064978.1065034

[11] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, any-time binary instrumentation. In Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools (PASTE '11). ACM, New York, NY, USA, 9-16. DOI=http://dx.doi.org/10.1145/2024569.2024572