Preventing Secret Leakage from `fork()`: Securing Privilege-Separated Applications

Umesh Shankar and David Wagner
{ushankar,daw}@cs.berkeley.edu
University of California at Berkeley

***Abstract*— If trusted processes' secrets or privileged system objects such as file handles are leaked to an untrusted process, the result could be the loss of secrecy and integrity of the data produced by the program. The advent of privilege-separated programs has led to an additional risk: sensitive data or system objects may be leaked when the trusted process of the privilege-separated application forks an untrusted child process. We have identified several channels by which information may flow to the child process: memory, the environment, memory mappings, filesystem information, and file descriptors. We propose fixes for each of these leaks. Most are handled by a static source code analysis of the target privilege-separated application's source code, but some require modifications to the kernel or compiler.**

As a proof of concept, we applied our technique to privilege-separated OpenSSH running on the Linux 2.6 kernel. Using our tools, we were able to verify easily that it does not leak secrets from its trusted components to its untrusted components; all sensitive data is erased or downgraded appropriately before being inherited by untrusted components. This suggests that our method is a useful way of reasoning about privilege-separated programs.

I. Introduction

A. Motivation

Trusted processes on a system frequently contain secret information, such as cryptographic keys or passwords, that could be used to compromise the secrecy and integrity of the process' data. In addition, since these processes are often granted extensive privileges, they possess capabilities like file handles and shared memory handles that could be used for privilege escalation. Keeping sensitive data and objects from leaking to an adversary is essential for trusted processes to maintain the secrecy of the application's output. Smith has also discussed the need to preserve secrecy to maintain output data integrity in his work on "outbound authentication" [16].

Noticing that many compromises of trusted applications came from the portions that handle user interaction, Provos et al. introduced *privilege separation* [11]. Privilege-separated designs separate the portion of an application that needs high privilege into small processes that offer a narrow interface to larger, unprivileged processes. This approach minimizes the code that must be privileged and guards it with a narrow interface, reducing the risk of privilege escalation; if the untrusted portion is compromised, the trusted portion still maintains its relative isolation.

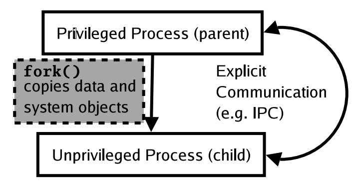

Fig. 1. **The question we consider in this paper** is whether any secrets or privileged system objects are leaked across the `fork()` interface from the trusted parent process to the untrusted child process; secret leaks may also compromise the parent's integrity. We ignore secrets passed across explicit channels, such as IPC or other communication.

There is a potential problem with privilege separation: since the untrusted processes are forked from the trusted ones, secrets may be leaked via the memory, environment variables, and other system objects that are copied on `fork()`. Unlike ordinary system output functions which explicitly accept all affected data in their arguments, `fork()` automatically copies all data and many system objects. Therefore, we need a systematic way to ensure that secrets are not leaked from the privileged component to the untrusted one because of `fork()`'s implicit sharing.

B. Our approach

We first identify the types of leaks that are possible (Section II) by examining the Linux kernel source code that implements the `fork()` system call. The set of copied data is summarized in Figure 2. We analyze each of these for its risks and propose solutions for each threat we identified. Some solutions involve explicitly discarding privileged objects in the child process, since these objects can be enumerated easily. However, identifying potential data leaks requires a more sophisticated approach because data is easily and frequently copied and used in the application. To find data leaks, we propose a new static analysis, combining control-flow (program slicing) and dataflow (type inference) analyses.

Some privilege-separated programs may call `exec()` from within the child process. The `exec()` system call clears the process' memory space and loads in a new binary. The clearing of the memory space decreases the number of potential leaks. However, using `exec()` can make data sharing between parent and child process more

Data/Object copied on **`fork()`**	Erased on **`exec()`**?	Notes
Code memory, sighandlers, pending signals	Yes	
Data memory	Yes	
Stack memory	Yes	
File descriptors	No*	*File descriptors with `CLOSE_ON_EXEC` flag are closed on `exec()`
Filesystem information	No	Includes filesystem root, filesystem namespace, umask, working directory
Process memory mapping	Yes	Includes `mmap()` regions
Shared memory segments	Yes	
Environment	No	Environment may be changed with `execve()`

Fig. 2. **Copy semantics of `fork()` and `exec()`.** We analyzed the Linux 2.6 kernel source code for `fork()` and `exec()` to see what data or system objects are copied from the parent to the child process by `fork()` and which are erased or closed by a subsequent `exec()`.

difficult by requiring more complex marshaling. This complexity can make it harder to verify the correctness of the (ideally very small) trusted process. Thus, whether or not `exec()` is used, we want some assurance that sensitive data and privileged system objects are not leaked during creation of the unprivileged child process. We discuss the details of achieving such assurance for both cases in the next section.

In this paper, we confine ourselves to preventing leakage during process creation. We do not address side-channel attacks or other observation mechanisms that may reveal secret information; we do not consider control dependences on sensitive data, just data dependences. Leakage to other processes or I/O channels via system calls such as `write()` is the subject of ongoing work. Furthermore, we assume that the unprivileged, untrusted child process has had its security context (typically UIDs and GIDs) appropriately downgraded.

As a proof of concept for our approach, especially the static analysis, we have applied our technique to OpenSSH and were able to show, with little manual effort, that it does not leak sensitive data to its untrusted portion. Our advances in supporting privilege separation are particularly significant in light of current efforts [3], [8] to simplify and automate privilege separation, which will likely speed adoption of the technique.

C. *Contributions*

In summary, we make the following contributions:

- We identify new risks to privilege-separated programs resulting from implicit copying of data and system objects from the privileged, trusted parent process to an unprivileged, untrusted helper child process by the `fork()` system call (Section II). During this copying, sensitive data and privileged system objects may leak to the child process.
- We propose a new method of systematically reasoning about such leaks (Sections II and III)—the first, to our knowledge—and propose fixes for each.
- We show that many such leaks may be detected using our proposed program analysis, which combines control-flow and dataflow analyses (Section III).
- We provide evidence that our method is easy to use by analyzing the most prominent privilege-separated application, OpenSSH (Section IV). Using our method, we showed that OpenSSH is free of sensitive data leaks.

II. IDENTIFYING POSSIBLE LEAKS

We examined the Linux 2.6 kernel's implementation of the `fork()` and `exec()` system calls to determine which data and system objects are copied from the parent process to the child, and which are erased if `exec()` is called afterwards. The results are summarized in Figure 2. Each line of the table represents a potential leakage channel. In this section, we analyze them to decide which represent real vulnerabilities. In our discussion, we assume that the untrusted child process has its user and group IDs or other relevant security context set correctly to reflect its untrusted status.

If a fine-grained access control system, such as SELinux [12], is present, some of these channels may be mediated by the system's security policy. SELinux is a reference monitor for Linux that allows fine-grained control over the use of system objects like files, file descriptors, and pipes. The system policy specifies which processes may use which types of system objects in which ways. Using SELinux can help prevent some kinds of leaks by interposing an access control check on the use of the leaked system object. We make a note of several ways that using SELinux can help prevent leaks.

A. *Enumeration of possible leaks*

We list all the relevant data and system objects implicitly copied by `fork()`, and analyze the risks associated with each as well as fixes for each potential leak. Note that some IPC mechanisms, such as System V Message Queues, perform an authorization check for each operation, so the lack of privilege in the child process serves to prevent privilege escalation. These are not considered here.

- **Code memory, signal handler table, and pending signals.** We do not need to trust the code of the child process—whether called from a signal or not—since it runs with low privilege. The code memory and signal handler set of the parent are not generally considered secret and may usually be derived from the source code; nonetheless, we can check that no sensitive data is used in setting up the signal handler table. We assume here that no secrets are embedded in the code of the parent process.
  Fix: Use our static analysis (Section III) to make sure that the signal mask and the set of signal handlers are not derived from sensitive data.
- **Data memory and stack.** Secrets could be leaked in data or stack memory copied to the child process. Also, there is an additional vulnerability whereby previously-used stack memory could leak secrets. Details are discussed below in Section II-B. If `exec()` is used, then the memory space is erased and no data or stack memory leak is possible.
  Fix: If `fork()` is used without `exec()`, use our static analysis (Section III) to detect sensitive data and stack memory leaks. The previously-used stack memory risks can be alleviated with kernel and compiler modifications (Section II-B).
- **File descriptors.** The parent process may have open file handles that should not be accessed by an untrusted child process, because opening them requires higher privilege than that granted to the child.
  Fix: Immediately after `fork()`, close all file descriptors in the child process except those needed for explicit communication with the parent. If `exec()` is called in the child, the `CLOSE_ON_EXEC` flag may be set on all unneeded descriptors by the parent process instead. These changes eliminate file descriptor leaks, but require programmer modification to the source code. SELinux may be useful in preventing this leak without source code changes. The default SELinux policy does not allow a child process to use the parent process' file descriptors, so by default no leak is possible. Any file descriptors required for communication may be explicitly allowed in the policy.
- **Filesystem information.** The main pieces of information in this data structure are the process' filesystem root directory, filesystem namespace (VFS mounts), the umask, and the current working directory. This information is not erased on `exec()`, but is not generally considered to be secret: in fact, it is visible to other processes via the `/proc` filesystem.
  Fix: None needed.
- **Process memory mapping.** The memory mapping itself is not a potential secret leak—indeed, it is visible on `/proc`—but the mapping may contain memory-mapped files which require privilege to access. Memory-mapped files are unmapped if `exec()` is called.
  Fix: If `fork()` is used without `exec()`, all memory-mapped files should be unmapped, except those needed for explicit communication with the parent. A list of such mappings is available in the `/proc` filesystem under `/proc/<pid>/maps`.
- **Shared memory.** Privileged shared memory regions could be abused by the untrusted child process if the parent process has attached to the region using `shmat()` before calling `fork()`, since the descriptor is copied on `fork()`. Shared memory descriptors are detached on `exec()`, so there is no risk of accidental leakage in that case.
  Fix: If `fork()` is used without `exec()`, a parent process may create shared memory regions using `shmget()` but should defer attaching to them using `shmat()` until after the `fork()` call. Our static analysis (Section III) checks that `shmat()` is not called before `fork()`. If attaching to shared memory segments before `fork()` is absolutely required by the application, the parent process should detach all shared memory segments immediately before calling `fork()`. A list of all shared memory mappings is available in the `/proc` filesystem under `/proc/<pid>/maps`.
- **Environment.** This information is not erased on `exec()`. Thus, if sensitive data is stored in environment variables, it should be erased immediately following the call to `fork()` even is if it is followed by `exec()`.
  Fix: Use our static analysis (Section III) to detect leaks of sensitive data into the environment. Alternately, the `execve()` system call allows explicit specification of the child's environment variables, avoiding accidental leakage.

Given that several fixes require data scrubbing or sanitization immediately before or after `fork()`, one may imagine a user library that implemented these together in a `secure_fork()` function. Such a function might detach all shared memory regions, call `fork()`, then, in the child process, unmap all memory-mapped files and close all open file handles.

B. Previously-used stack memory leaks

The stack is inherited across `fork()`. This means that there is a potential information leakage on the stack from leftover stack variables. Consider the example in Figure 3. If secret data is on the stack in a local variable, and subsequent stack frames before `fork()` are all at higher memory addresses, the secret data may be left in unused stack memory in the new child process. To avoid this problem, the operating system should zero-fill stack memory in the child below the current stack pointer immediately after any `fork()`. Note that this change might break code which relies upon the unsafe practice

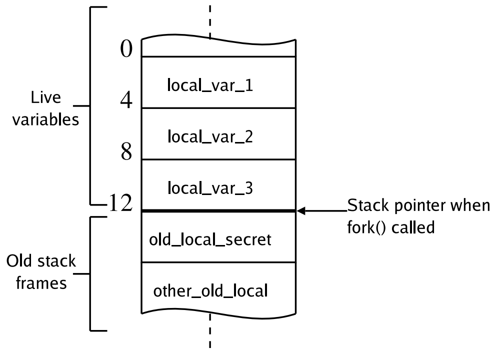

Fig. 3. **Previously-used stack memory can leak secrets.** If a secret is in a stack location which is never overwritten before `fork()`, a leak can result. Here, `old_local_secret`'s contents may remain in unused stack memory when `fork()` is called. The operating system should zero-fill all memory below the stack pointer in the newly created child process.

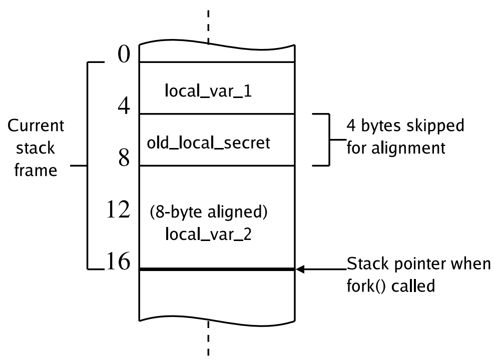

Fig. 4. **Alignment restrictions can leak secrets.** If a secret is in a stack location which is not overwritten due to alignment constraints, a leak can occur even if the stack is zero-filled below the stack pointer. Here, `old_local_secret`'s contents may be on the stack from a previous stack frame when `fork()` is called because `local_var_2` must be aligned to an 8-byte boundary.

of using the address of a local variable from a function that has already returned. Since such code is already likely to fail, as it is not memory-safe, we consider this to be an acceptable tradeoff.

Even if the zero-filling recommendation above is adopted, it is still possible for data to leak on the stack due to machine- and compiler-specific alignment options. Consider the example in Figure 4. An 8-byte aligned variable (`local_var_2`) may appear in the source code to overwrite a secret, since the size of the current stack frame is large enough to overwrite the previous frame containing the secret. However, the alignment restriction leads to a leak since the bytes containing the secret are skipped. To avoid this problem, compilers should make sure that all extra bytes needed for alignment or padding are zero-filled.

III. Static Source Code Analysis

In Section II, we proposed a static source code analysis to detect certain kinds of data leaks. Here, we describe that analysis. In particular, the property we want to check is that all data that is live at the time `fork()` is called, including data memory, stack variables, environment variables, and attached shared-memory regions, is either not sensitive or is immediately downgraded after `fork()`.

A. Overview of the analysis

Checking the desired property requires five steps:

1) The programmer annotates a small set of variables in the program as containing sensitive data (Section III-C).
2) Our algorithm automatically computes the set of live data by using a *program slice* (Section III-D) that captures the code executed before `fork()`, including the active stack frames when `fork()` is called. All stack variables in active stack frames, as well as those passed to `setenv()` (are thus are in the environment), are marked as live. Global variables are always considered live.
3) The algorithm then infers the set of derived sensitive data in the program slice from the programmer annotations, using the procedure described in Sections III-C and III-E.
4) The algorithm intersects the live and sensitive sets. Data that is both live and sensitive represents a potential leak, and the algorithm reports the leak as a warning message.
5) If there are any warning messages, the programmer checks that the data in question is scrubbed immediately after `fork()` in the unprivileged child process. In addition, if the `shmat()` system call, which attaches to a shared memory region, is called before `fork()`, the system reports this as a possible leak, and the programmer should rewrite the offending code to defer attaching to the region until after `fork()` is called.

B. Tools used

In order to perform our analysis, we employ the Oink/Elsa front end [9], [10] analysis framework. This framework provides a C/C++ parser and facilitates performing analyses on the resulting abstract syntax tree (AST). It also includes integration with multiple backend analyses. We extended this backend interface to work with the fully polymorphic (context-sensitive) CQual type inference system backend [7]. CQual performs sound type inference on built-in and user-supplied qualifiers in C programs, and is well-suited to a dataflow analysis [2], [13], [17], described in more detail in Section III-C. Using the Oink/Elsa framework, we were able to implement our program slicing on the AST, then make automatic annotations to the tree based on the slice, before invoking the CQual backend.

C. Dataflow analysis using type inference

A *type inference* analysis may be used to trace the flow of various kinds of marked data in a program, and to enforce restrictions on data with certain annotations, called *type qualifiers*. CQual is a system that allow programmers to add type qualifiers with associated rules to C programs, and has been used for similar applications including finding format-string bugs [13] and isolating sensitive data in an application's memory space [2]; the reader is referred to these for a more thorough exposition of the application of type inference to security than is provided here. In our case, we want to be able to infer all potentially sensitive data from a small set of sensitive data that has been so marked by a programmer, then flag as a potential leak any potentially sensitive data that is live when `fork()` is called.

Say the program uses a secret key, which it stores in a variable, `key`. The program might then copy the key or generate a subkey from it for use in a crypto algorithm:

```
Key key, key_copy, subkey;
...
key_copy = key;
make_subkey(key, &subkey);
```

Now, we do not want to have to find all the data derived from the secret key by hand; we would like to indicate that the variable `key` is sensitive and have an analyzer infer all other sensitive data from it. CQual lets us mark the secret key as being sensitive using a special type qualifier, `$sensitive`:

```
$sensitive Key key;
```

Type qualifiers also come with rules: a pointer to a `const` object may only be assigned to another pointer to `const`, for example. Likewise, we may also specify rules about how the `$sensitive` qualifier propagates, that is, how it is inferred to apply to other variables. Here, we specify that all variables either derived from a sensitive one, or from which a sensitive one is derived, are marked sensitive. The inference system then infers the qualifiers on other variables:

```
$sensitive Key key;
$sensitive Key key_copy, subkey;
...
key_copy = key;
make_subkey(key, &subkey);
```

It is easy to see that `key_copy` should be `$sensitive`, since it is directly assigned from `key`. However, CQual is capable of looking inside the `make_subkey` function and determining that `subkey` is derived from `key` as well, thus correctly inferring the `$sensitive` qualifier on `subkey`.

Finally, we may specify rules governing the interactions between qualifiers, just as C has rules for its built-in qualifiers, e.g., no `const` variable may be assigned from a `volatile` one. In particular, the set of variables that are live at the time of `fork()` are annotated as `$live` by our analysis (see Sections III-D and III-E). We add a rule indicating that any variable inferred to be both `$sensitive` and `$live` should generate a warning message, as it represents a potential leak. Section III-E shows in more detail how to tie the slicing and inference together for leak detection; for a more complete example, see Figure 5.

D. Program Slicing

a) Slicing Algorithm: In order to determine which variables to mark as `$live`, we must perform a control-flow analysis to decide which code is executed prior to `fork()` and which stack frames are live when `fork()` is called. This requires generating a *program slice* that contains just that set of code.

Generating a straightforward flow-independent backward slice from `fork()` would likely end up including the whole program, since the parent and child differ only in the return value of the `fork()` function. The reader is referred to Tip's survey of program slicing techniques [14] for more details about difficulties with standard slicing algorithms.

Instead, our algorithm makes some reasonable assumptions about the context in which `fork()` is called in order to generate a correct slice in the common case; in Section III-D.0.b, we justify our choice. A more conservative approximation, which might yield more false positives, would be simply to perform the type analysis on the whole program instead of using a slice.

Our algorithm consists of the following steps:

1) From the AST, build a callgraph for the program. Use a conservative alias analysis to include calls through function pointers.
2) Define the set T to be the transitive closure, going backwards from `fork()`, of the set of functions that call `fork()`.
3) Starting in `main()`, do a depth-first search of the callgraph, stopping when a call to `fork()` is reached. Add each function along the way to the set F. The depth-first search approximates the usual order of execution, starting in `main()`. For completeness, we repeat the procedure, starting in each function that is used as a signal handler rather than starting in `main()`. Now F contains the set of functions that are (potentially) called before a `fork()`.
4) F is a slight overapproximation to the slice we want: if a function calls `fork()`, then no code in the

function following that call should be included in the slice. The same is true if a function calls a function $f \in T$: `fork()` would be called before f returns. We therefore refine our slice as follows. For each function in F, traverse the statements in order, marking each one encountered as reachable. Stop traversal after encountering a call to a function in the set T or when encountering a direct call to `fork()`.

5) For each signal handler, we search the callgraph to ensure that `fork()` is never called from it. Now the set of marked statements constitutes the program slice.

An example of the slicing algorithm is given in Figure 5, along with the computation of the T and F sets.

b) Correctness of the slicing algorithm: The slicing algorithm given above assumes that there is only one `fork()` call in the program, and that it is called at the first possible point (assuming all relevant conditionals are true). More precisely, the algorithm is correct if and only if all paths by which a `fork()` call may be reached are equivalent from a secrecy perspective.

If there is more than one program point at which `fork()` is called, we can simply analyze each separately by removing all but the target call and running the analysis.

A difficult case occurs when, for a given child process, the `fork()` point is gated by a condition such that the set of code that is executed before the `fork()` is unclear. Consider this example:

```
while (...) {
  pid = fork();
  if (pid == 0) {
    child_code();
    exit();
  } else if (pid > 0) {
    parent_code();
  } else if (pid == -1) {
    error_handling_code();
  }
}
```

The trouble is that our slicing algorithm cannot easily distinguish between `parent_code()`, which is always executed in the parent *after* `fork()`, and `error_handling_code()`, which is executed after a failed attempt to fork, but *before* a successful attempt. The consequence is that the error-handling code will be wrongly excluded from the slice. If this code affects which secrets are live, we could miss real security holes.

Fortunately, this does not seem to be a problem in practice: error handling code tends not to manipulate sensitive data or objects. We looked at a sampling of popular daemons to see how they called `fork()`. All of them, including OpenSSH, the Apache HTTP server, the Cyrus IMAP daemon, and the WU-FTPD FTP server, use some variant of the idiom above. If, however, the position of the `fork()` call were not secret-independent, say if additional values got assigned secret data in `error_handling_code()`, an extension to our algorithm would be required. We have not yet found any program that would necessitate such an extension.

We check that `fork()` is not called nondeterministically from a signal handler. Note that although `setjmp` and `longjmp` also introduce exceptional control flow, since they respect the stack they do not expand the program slice and so are not a problem for our approach.

E. Finding leaks

There are three kinds of possible leaks detected with the static analysis: attached shared memory regions, data and stack memory secrets, and secrets in the environment. The first is simple to check conservatively: we simply look for shared-memory attach (`shmat()`) calls in the program slice. We have not yet implemented a more precise analysis that tries to match pre-fork detaches with attaches.

As for the latter two kinds, once we have the appropriate program slice, we can apply type inference as described in Section III-C. The set of live data at the time of `fork()` is conservatively defined to be: the set of global variables; all the local variables of any function that might be on the stack when `fork()` is called (functions in the set T in Section III-D, above); and all values passed to `setenv()`. These are automatically marked with a `$live` annotation following the program slicing operation. Recall that sensitive data structures are annotated once by hand with a `$sensitive` annotation. Using CQual, data derived from sensitive structures is also inferred to be `$sensitive`. CQual is configured such that any data that has both the `$sensitive` and `$live` qualifiers is flagged with an error message as being a potential leak. An example of the algorithm is given in Figure 5.

Once potential leaks have been identified, the programmer must ensure that all flagged data is erased or otherwise sanitized immediately after `fork()` is called.

F. Soundness of the analysis

Our static analysis algorithm is sound if it catches all the leaks due to shared memory regions, data memory, stack memory, and environment variables. The correctness of the program slice is discussed above in Section III-D.0.b. Our analysis on the program slice is as sound as CQual because of the conservative way in which we determine the set of live variables. An exception is if `fork()` is called from a signal handler, which would violate the usual control flow. We check for that condition while generating the slice. CQual itself is unsound in certain cases: inline assembly, dynamically generated code, and memory-unsafe code

Original

```
int $sensitive secret();
int f() {
  int i;
  return fork();
}

void h() { ... }

void g(int s) {
  int n = s;
}

int main() {
  int m = secret();
  g(m);
  pid = f();
  h();
}
```

+ slicing

```
int $sensitive secret();
int f() {
  int i;
  return fork();
}


void g(int s) {
  int n = s;
}

int main() {
  int m = secret();
  g(m);
  pid = f();

}
```

+ inference

```
int $sensitive secret();
int f() {
  int $live i;
  return fork();
}


void g(int $sensitive s) {
  int $sensitive n = s;
}

int main() {
  int $sensitive
      $live m = secret();
  g(m);
  pid = f();
}
```

The set T from Section III-D for this example program is {`f()`, `main()`}, since `f()` calls `fork()` and `main()` calls `f()`.

The set F from Section III-D for this example program is {`main()`, `secret()`, `g()`, `f()`}, since each of these functions is called before `fork()`. Note that the actual slice does not include all of `main()`: since `f` $\in T$, `fork()` is called before `f()` returns to `main()`. Therefore, the code following the call to `f()` in `main()`, namely the call to `h()`, is omitted.

Fig. 5. **Verifying the secrecy of live data.** The original code, with a single `$sensitive` user annotation, is shown on the left. The middle column reflects program slicing to eliminate code called only after `fork()`. It eliminates the function `h()` and the code in `main()` following the call to `f()`. On the right, variables that are live at the time of `fork()` (`m` and `i`) are automatically annotated with `$live`. Type inference is performed. Variables that contain sensitive data (`m`, `n`, and `s`) are then inferred to be `$sensitive`. Data leaks are indicated by variables marked both `$sensitive` and `$live`. In this case, m represents a potential leak.

(such as the presence of buffer overflows). Our analysis does support inference through function pointers.

IV. Experiment: Analysis of OpenSSH

We applied our static analysis to the OpenSSH 3.9 daemon to see if there were any sensitive data leaks from the privileged processes to the unprivileged ones. In our tests, we analyzed OpenSSH along with the OpenSSL library that it uses for many cryptographic functions; these were combined into a single file for analyses using the CIL tool [5]. The source we analyzed consisted of 1.2 million lines of code. Our tests were performed on a 64-bit 800-MHz Itanium CPU with 13GB RAM, running Red Hat Enterprise Linux with a the Linux 2.4.21 kernel and gcc 3.2.3. Running our static analysis required 420 minutes and 4.8 GB RAM.

Privilege-separated OpenSSH consists of several processes. One privileged component, called `listen`, listens for new connections and spawns a separate, trusted process called `priv` for each new connection. This `priv` component performs the per-connection privileged operations required by OpenSSH: authentication of the remote user, creation of pseudo-terminals, transition to a particular, authenticated userid, and session traffic encryption and decryption. The `priv` component in turn spawns unprivileged components to handle various types of user interaction. The `net` component is used to perform the remote interaction part of the authentication phase, which has in the past been subject to compromise; it uses `priv` as a privileged server via a narrow interface. After successful authentication, `priv` spawns a shell or other process requested by the user in that user's security context.

We applied our static analysis (Section III) for each of two child process creations: `priv` creating the `net` process and the user shell process. The user shell process is created using `exec()`, so we only needed to consider a subset of our analysis results (see Section II for details). The `listen` process that spawns `priv` is also trusted, so we did not analyze that transition.

OpenSSH is a well-designed system, and isolates all its sensitive data into a single global structure, `sensitive_data`. We annotated this data structure as being sensitive and proceeded with the algorithm.

Despite the large number of live variables at the time of `fork()`, only a single variable in addition to `sensitive_data` was flagged as containing a potential leak in the creation of `net`. We hand-verified that the contents of `sensitive_data` are downgraded or destroyed in unprivileged child processes immediately following each `fork()` transition. This was straightforward since we the amount of code to be examined—the code that followed `fork()` directly—was small.

The additional variable identified by the analysis, a pointer named `key`, was a local variable in the main()

function that was used to read secret keys from disk and initialize the sensitive data structure. Here is a code snippet where it was used:

```
Key *key = key_load_private(...);
sensitive_data.host_keys[i] = key;
```

Because of the way `key` was used, all the values it pointed to were also pointed to in sensitive data structure. Those values were destroyed or downgraded suitably when the sensitive data structure was erased; this fact was hand-verified using a debugger. One can easily imagine that if the data were copied by value to the sensitive data structure rather than by reference, a real leak would have been present.

In the creation of the user shell process, we only needed to consider data leaks via the environment, since it used `exec()`. Our static analysis found that no such leaks exist.

V. Related Work

Static analyses using the CQual type inference system, like the one we perform in Section III, have been applied to security before. CQual has been used to find format string vulnerabilities [13], verify the correctness of authorization hooks in the Linux kernel [17], and generate secure crash information [2]. More recently, a fully polymorphic (context-sensitive) version of CQual was introduced by Johnson and Wagner to find user-kernel bugs [7]; we use this version of CQual in our analysis.

Static analysis has recently been applied to the problem of automating privilege separation in a system called Privtrans [3]. Another effort to ease the implementation of privilege separation is the Privman library [8]. If automated privilege separation becomes more usable, then we expect that more systems will adopt this approach. If so, our analysis, which is the first to support privilege separated applications in addition to unseparated ones, will be increasingly useful.

Recently, Chow et al. used hardware-level simulation on a virtual machine in order to perform a dynamic cross-process taint analysis [4]. Our work focuses on using static analysis rather than dynamic tracing. Our approach does not suffer from the code-coverage problem of dynamic analysis; conversely, the dynamic approach sometimes has fewer false positives and is good at tracking data across applications, which our tools do not currently support.

VI. Conclusion

The advent of privilege separation as a technique to reduce the TCB of security-critical applications has brought with it some new security risks, since data and system objects are automatically copied when a privileged process spawns an unprivileged one using the `fork()` system call. In this paper, we have systematically identified the ways in which sensitive data and system objects may leak from a trusted parent process to an untrusted child. We have proposed fixes for each potential leak, including a static source code analysis in order to catch many kinds of data leaks. Finally, we used our techniques to prove that OpenSSH does not leak secrets to its untrusted components, at the same time demonstrating the practicality of our approach.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[2] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2003

[3] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, August 2004.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, August 2004.

[5] G. Necula, S. McPeak, S. Rahul and W. Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In *Proceedings of the Conference on Compiler Construction*, 2002.

[6] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb 2002.

[7] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, August 2004.

[8] D. Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, April 2003.

[9] S. McPeak, G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proceedings of Conference on Compiler Construction* (CC04), April 2004.

[10] Oink program analysis framework. Available from `http://freshmeat.net/projects/oink/`.

[11] N. Provos, M. Friedl and P. Honeyman. Preventing privilege escalation. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, August 2003.

[12] National Security Agency. Security-Enhanced Linux (SELinux). `http://www.nsa.gov/selinux`, 2001.

[13] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. "Automated Detection of Format-String Vulnerabilities Using Type Qualifiers," in *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[14] F. Tip. "A survey of program slicing techniques". In *Journal of Programming Languages* 3(3) , (1995), 121-189.

[15] S. Smalley. "Re: fork and security context transitions" on SELinux Mailing List, 3 Feb 2004. Available at `http://www.nsa.gov/selinux/list-archive/0402/6391.cfm`.

[16] S. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of the* 8^{th} *European Symposium on Research in Computer Security (ESORICS)*, 2002.

[17] X. Zhang, A. Edwards and T. Jaeger. "Using CQUAL for Static Analysis of Authorization Hook Placement". In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.