USENIX Association

Proceedings of the 12th USENIX Security Symposium

Washington, D.C., USA
August 4–8, 2003

Scrash: A System for Generating Secure Crash Information

Pete Broadwell Matt Harren Naveen Sastry*
University of California, Berkeley
{pbwell, matth, nks}@cs.berkeley.edu

Abstract

This paper presents Scrash, a system that safeguards user privacy by removing sensitive data from crash reports that are sent to developers after program failures. Remote crash reporting, while of great help to the developer, risks the user's privacy because crash reports may contain sensitive user information such as passwords and credit card numbers. Scrash modifies the source code of C programs to ensure that sensitive data does not appear in a crash report. Scrash adds only a small amount of run-time overhead and requires minimal involvement on the part of the developer.

1 Introduction

Developers often examine a failed program's state to diagnose and fix software bugs. For this reason, operating systems and programming suites include tools to capture a program's state in a core file at crash time. The recent advent of ubiquitous network connectivity for personal computers makes *remote crash reporting* possible, whereby programs send crash information back to developers after a failure. This practice, which allows developers to receive information about bugs in their programs after the programs have been distributed to users, has since become commonplace [1, 2, 3, 4]. Remote crash reporting offers many readily apparent benefits to developers, but the privacy-related implications of the technology are not as well understood.

The contributions of this paper include

- An explanation of the privacy-related problems posed by remote crash reporting.
- An analysis of the tradeoffs inherent in the design of core file cleaning systems.
- A description of an easy-to-use system that safeguards the user's private information from exposure via crash reports.

*The authors were supported in part by DARPA NEST contract F33615-01-C-1895, NSF grants CCR-0093337, CCR-0085899, CCR-0085949, and CCR-0081588, and a grant from Microsoft.

1.1 Crash reporting background

Remote crash reporting tools appear in many forms, but all perform the same basic task: gathering and transmitting crash-related data to a remote database. Microsoft's Dr. Watson tool for Windows [4] performs crash-reporting services for most Microsoft applications. Another Windows-based tool is BugToaster [2], a third-party crash data collection utility that sends its reports to an independent database. Bug-Buddy for the GNOME desktop environment [1] and Mozilla Talkback [3] are remote crash reporting tools used in the open source community.

The types of data contained in a remote crash report can vary widely, depending upon the configuration of the reporting tool. Crash reporting tools typically record some information about the environment in which the failed program was running, including the error associated with the crash, program version information, loaded drivers, memory usage and open files. They also send back a subset of the data present in a core file: part or all of the call stack, processor registers, and heap contents of the crashed program.

Remote crash reporting technology grants the developer access to potentially vast amounts of crash data, speeding the diagnosis and repair of software vulnerabilities. For example, developers can fingerprint the call stacks returned in crash reports to determine which bugs appear most often and thus deserve the most attention. The developer also can suggest fixes or patches to the user based on the contents of a crash report.

Given the increasing prevalence of remote crash reporting, it is important to consider the security-related risks associated with the technology. Because they contain some or all of the memory contents of the program at the time it failed, crash reports may include sensitive user information such as credit card numbers, passwords or web browser cookies. A recent Department of Energy security advisory regarding the Dr. Watson crash report tool for Office XP and Internet Explorer warns that the program could send sensitive information to Microsoft, since the memory dump in the crash report might contain portions of the document being viewed [12]. A related concern about Dr. Watson is that the program stores comprehensive crash reports in a world-readable directory on the host computer [18]. This practice raises security and privacy concerns, because a malicious party on a multi-user system could examine the crash report and extract confidential information.

There are inherent privacy risks associated with sending crash data to a remote party over a network. An initial vulnerability is that the data may be intercepted en route. Dr. Watson guards against this threat by encrypting the data stream with SSL [4], but GNOME's Bug-Buddy currently sends crash reports unencrypted via `sendmail` [1].

The primary concern, though, is the fate of the failure data after it reaches the crash data repository. These repositories contain crash reports from many users, and may become popular targets if they are known to house sensitive information. An important distinction here is that the user trusts the developer to produce quality software – the user installs and executes the software voluntarily, after all. The user should not, however, be obligated to trust the developer to safeguard his sensitive crash data for an indefinite length of time. Securely maintaining data takes a different kind of expertise than writing secure and correct code. Thus, users may be more willing to participate in remote crash reporting if the crash reports can be stripped of personal information.

We also hypothesize that developers often won't want to store a user's sensitive information. The inclusion of privacy-sensitive information in the crash report presents a risk for the developer: a security breach of a crash repository could result in bad publicity or financial liability. For these reasons, we believe that *both* users and developers would like to eliminate sensitive information from crash data.

2 Core File Filtering Systems

A *core file* is a snapshot of a program's execution state generated when a crash occurs. We propose a core file *filtering system* as a method of identifying *sensitive information* and ensuring that it does not appear in a core file. The developer decides which categories of data should be considered "sensitive" for each particular executable; the filtering system must then prevent sensitive information from appearing in the final crash report. Conversely, *insensitive* information is allowed to appear in the crash report. A filtering system is composed of two separate phases: the first phase transforms the application source code, and the second phase transforms core files that result from application crashes.

We place two restrictions on the source code modification phase: the behavior of the application to be modified must be indistinguishable from that of the original, and the transformation should not modify the program in a way that makes debugging the filtered core file unnecessarily difficult. Since the filtering system is supposed to preserve a developer's ability to debug the original application, the transformation must preserve the variables and control structure of the application to the greatest extent possible. For example, we allow transformations that move the memory locations of variables since the contents of these variables are still present in the resulting core file. Thus, an *information-preserving* source code transformation retains all of the same variables of the original program but may rearrange their layout in memory.

The second phase of a filtering system modifies the core file generation process so that no sensitive data appears in the core file. In practice, this task can be accomplished by running a separate program to delete selected information from a complete core file after it has been generated.

We now outline two metrics to characterize the effectiveness of the filtering system. The first metric measures the usefulness of the core file to the developer, since debugging a crash is more difficult if a critical piece of data has been removed from the core file. Using this metric, the original, full core file is the most useful for debugging, while an empty core file is useless. The second metric measures the filtering system's effectiveness from the user's perspective, i.e., how well the system protects the user's privacy and data. Using this metric, a user's privacy is best preserved if the filter removes all information.

The challenge, then, of designing a filtering system involves balancing the needs of the developer with those of the user. The filtering system must preserve as much information as possible for the developer while maintaining privacy for the user. A developer may choose any number of different privacy guarantees, depending on the particular application and the degree to which privacy is necessary. One such guarantee, for example, may prevent passwords from being leaked, but may not conceal the length of the password if this value is useful for debugging.

This model assumes that the developer is trustworthy. It does not guard against privacy violations by malicious developers, since a developer can easily insert a covert channel into the program. Rather, the developer controls the filtering system and defines the balance between the user's privacy and the developer's need to debug the application. We imagine that advanced filtering systems might even give the user a choice between multiple privacy-utility tradeoffs. Thus, the primary goal of a filtering system is to protect against privacy violations after the core file has been generated, particularly in crash repositories.

2.1 Scrash goals

Our system, Scrash, is an easy-to-use filtering system that presents several tradeoffs between privacy guarantees and developer utility of crash data. Its goal is to eliminate sensitive memory locations and their copies from a core file. In addition, Scrash provides developer control over certain classes of derivative data that may be removed from the core file. For example, Scrash considers the length of a sensitive buffer to be sensitive as well, which ensures that the length of a sensitive password buffer computed via `strlen` will also be regarded as sensitive. The developer may choose to override this rule, however, if she feels that disclosing the length of the buffer may be beneficial for problem debugging and does not pose a significant privacy risk.

Scrash ignores privacy leaks resulting from indirect information flows or other covert channels. As an example of such an information flow technique, the program counter and call stack can leak information on the state of sensitive variables. Consider the following example:

```
char c = password[0];
if (c >= 'a' && c <= 'z') {
  // stmt a
} else {
  // stmt b
}
```

If the program's execution state indicates that statement *b* was executed, then an adversary can infer that the password does not start with a lower case letter even if the password variable is marked as sensitive. Eliminating control flow privacy leaks and other covert channels while retaining enough information for debugging is difficult, so Scrash ignores such vulnerabilities. For example, the processor registers and even the entire call stack would not be available to the developer in a system that seeks to guard against control flow privacy leaks. All reveal the state of prior control flow decisions and could be used to discover information about the state of sensitive variables that had been used in conditionals.

3 Implementation

Scrash seeks to eliminate sensitive information from the heap, stack, and global variables while still providing useful information to the developer. We perform source code transformations to place the contents of any sensitive variables in a separate region of memory, which we then erase during core file generation to ensure that it is not transmitted as part of a crash report. Thus, the stack, globals and main heap in our modified core file will only contain insensitive information, so that the crash reporting tool is free to transmit any of these regions. The key difficulty of this task, which we will address below, is identifying the sensitive data. Even though the heap is not often transmitted using current crash reporting tools, we make a distinction between the sensitive and insensitive heap in the case that it may be transferred when sending a more detailed crash report. Making this distinction has a negligible performance cost, so we view the added safety it provides as worthwhile.

We implemented the source code transformation phase in 1200 lines of new Objective Caml code. We link the modified application with a memory allocator to which we added 250 lines of new C code. We wrote the cleaning phase using 90 lines of C code.

3.1 Merging of source files

We use CIL (a C Intermediate Language implemented in OCaml) [11] as the infrastructure for our source-to-source translation. CIL translates C code into a clean, easy-to-manipulate subset of C. It includes drivers that act as drop-in replacements for `gcc`, `ar`, and `ld` so that CIL can be used with existing makefiles. CIL uses these drivers to collect all of the source files for a program, preprocess them, and merge them into a single C file to facilitate whole-program analysis.

3.2 Analyzing the sensitivity of variables

Our system extends each type in a C program with a *type qualifier* to indicate whether or not it may hold sensitive information. Type qualifiers are an additional specification of the traditional C types. For example, "`$sensitive int`" is the type of an integer variable that may hold sensitive information at some point during its lifetime. When declaring a variable, the developer can specify that the variable will contain sensitive information by adding the `$sensitive` annotation. For all unannotated variables, we use the CQual type qualifier inference engine to determine whether the variable may hold sensitive information [14].

CQual performs an interprocedural program analysis to determine where sensitive data might flow from the initial set of sensitive variables annotated by the programmer. If CQual detects an assignment from a sensitive variable to an "unconstrained" variable, the unconstrained variable will be considered sensitive. Thus, CQual determines where the `$sensitive` qualifier spreads throughout the program. After CQual has finished, we know that all remaining unconstrained variables only contain insensitive data, since they never receive any assignments from sensitive variables. Conversely, if CQual determines that a variable is sensitive, it may contain sensitive information during the execution of the program, since there is a possible assignment to it from a known sensitive variable. The question of whether data may be sensitive is analogous to the question of whether it may be *tainted*, so we can use the same analysis as in Shankar et al. [14].

As an alternative to annotating specific data at the point it enters the program, the programmer may choose to use a pre-annotated header file that marks as sensitive all data returned by functions like `read` and `recv`. At the cost of unnecessarily marking some values as sensitive, this option makes it easy to denote user data as sensitive without the need to enter program-specific annotations. We take this approach in our evaluation experiments.

The CQual stage outputs the original program with attributes added to each variable describing its sensitivity. These annotations allow later stages of Scrash to determine whether a variable should reside in the secure or insecure region of memory.

3.3 Smalloc: secure malloc

After identifying sensitive variables, it becomes possible to erase their contents before shipping the core file. A difficulty arises in determining where the information resides in the core file, however, since in general the sensitive variables will be scattered throughout the entire core file. We need a way to communicate sensitivity information from the static analysis to the runtime cleaning process.

One method to recognize sensitive variables would be to append an immutable tag to each sensitive variable; the tag would describe the variable's sensitivity status. The post-processing cleaning step could then iterate over the core file and remove or overwrite all sensitive variables by checking for the tag.

An alternative approach, which we utilize, groups sensitive memory locations together and places an identifying header at the start of the region. This approach is ultimately more space-efficient than tagging each variable separately and simplifies the process of removing sensitive data from the core file.

We have written Smalloc, a region-aware memory allocator, to manage this "secure" region. It is based on the Vmalloc package, which provides an ideal platform for creating allocators [15]. The interface to Smalloc is similar to `malloc`. The only difference is that we add an extra parameter to the `smalloc` function to identify whether the new memory should be allocated in the sensitive or insensitive memory region. The signatures of the `realloc` and `free` functions remain unchanged. See Figure 2 for the complete Smalloc interface.

Smalloc creates sensitive memory regions for heap allocated variables, sensitive global variables, and the sensitive stack. We will discuss these regions below. Each of the regions is actually embedded within the normal heap segment. The globals and stack regions are statically sized and allocated at program initialization. The size of the sensitive heap region is dynamic.

3.4 Transformations

We perform transformations on the program source code to ensure that the variables the CQual phase marks as sensitive are placed into the sensitive memory region by Smalloc library routines. CIL provides an ideal platform for performing these transformations. We outline each of the transfor-

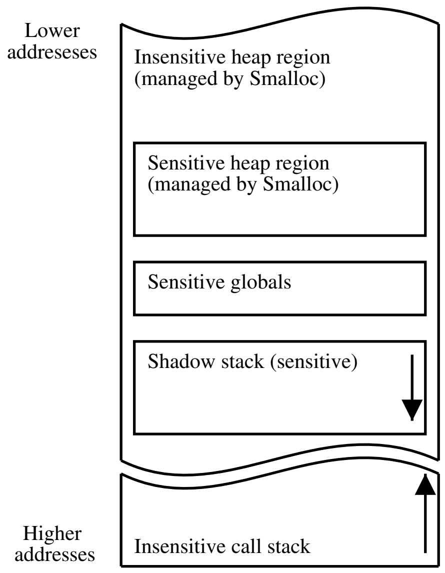

Figure 1: Layout of a process's memory when using Scrash. Both heap regions are managed by Smalloc. The sensitive globals, sensitive stack, and sensitive heap are embedded within the insensitive heap region. The arrows indicate the direction of stack growth.

```
void * smalloc (size_t size, int issecure);
void * scmalloc (size_t nmemb, size_t size, int issecure);
void sfree (void * ptr);
void * srealloc (void * ptr, size_t size);
```

Figure 2: The Smalloc allocator interface. The allocation functions take an extra boolean parameter that specifies whether the data should be allocated in the sensitive region or on the insecure heap.

```
#include <crypt.h>
#include <malloc.h>
#include <string.h>
int $sensitive private[2] = {0, 1};

void getPassword(char cryptpw[14]) {
  char $sensitive * password = malloc (255);
  memcpy (cryptpw, crypt (password, "00"), 14);
}

void check() {
  char $sensitive cryptpw[14];
  getPassword(cryptpw);
}
```

Figure 3: A sample code fragment that we will use to illustrate some of the transformations that Scrash uses (see Figure 4). It contains a sensitive global, a pointer to sensitive data, and a sensitive stack variable.

```
typedef unsigned int size_t;
struct check_shadow {
   char cryptpw[14] ;
};
struct __smalloc_globals {
   int private[2] ;
};
struct __smalloc_globals *__smalloc_global_var ;
void ( __attribute__((__constructor__)) __smalloc_global_init)() ;
char *stackPointer    =    0;
void *srealloc(void *ptr , unsigned int size ) ;
void sfree(void *ptr ) ;
void *scalloc(unsigned int nmemb, unsigned int size, int issecure ) ;
void *smalloc(unsigned int size , int issecure ) ;
extern char *crypt(char const *__key , char const *__salt ) ;
extern void *malloc(size_t __size ) ;
extern void *memcpy(void * __restrict __dest ,
                    void const *__restrict __src, size_t __n) ;
void getPassword(char *cryptpw ) {
  char *password ;
  char *tmp ;
  void const    * __restrict     tmp___0 ;
  {
    tmp = (char *)smalloc(255U, 1);
    password = tmp;
    tmp___0 = (void const *)
      crypt((char const *)password, (char const*)"00");
    memcpy((void *)cryptpw, tmp___0, 14U);
    return;
  }
}
void check(void) {
  struct check_shadow *check_shadow ;
  {
    check_shadow = (struct check_shadow *)stackPointer;
    stackPointer = stackPointer + sizeof(struct check_shadow );
    getPassword((char *)(check_shadow->cryptpw));
    {
      stackPointer = (char *)check_shadow;
      return;
    }
  }
}
void __smalloc_global_init(void) {
  {
    __smalloc_global_var = (struct __smalloc_globals *)
        smalloc(sizeof(struct __smalloc_globals ), 1);
    __smalloc_global_var->private[0] = (int )0;
    __smalloc_global_var->private[1] = (int )1;
  }
}
```

Figure 4: The results of running Scrash on the code fragment from Figure 3. Note that the constructor function `__smalloc_global_init` allocates the sensitive global inside the `__smalloc_globals` structure, which is allocated on the sensitive heap. This constructor function runs prior to `main` and is specified with the `__constructor__` attribute. The sensitive heap variable `password` is now allocated on the secure heap. Finally, the sensitive local array `cryptpw` is allocated on the shadow stack. A new structure, `check_shadow`, contains this variable. Maintenance of the shadow stack is performed on entry and exit of the `check` function.

mations below. The results of applying the complete set of transformations to the program in Figure 3 can be seen in Figure 4.

3.4.1 Sensitive heap variables

We allocate memory on the sensitive heap when the results of a `malloc` call are assigned to a pointer declared with the `$sensitive` qualifier. Recall that CQual assigns this qualifier to variables that could potentially contain sensitive information. Similarly, the absence of the `$sensitive` qualifier on a pointer indicates that the memory should be allocated on the insensitive heap. Thus, we change each of the allocation calls to use the Smalloc allocator, using the presence of the `$sensitive` attribute to denote which region the Smalloc allocator uses. We similarly replace `calloc` with `scalloc`.

In addition to replacing the allocation functions, we replace any instances of `free` and `realloc` with the Smalloc equivalents: `sfree` and `srealloc`. These functions have the same arguments and return types as the functions they replace, so we can perform a simple substitution.

3.4.2 Sensitive stack variables

There are two possible transformations that can be applied to place sensitive stack variables within the secure memory region:

Heap allocation of local variables. This transformation moves the sensitive stack variables into the secure heap. At function entry, we allocate a block of space on the secure heap for all of the sensitive local variables, which we deallocate before exiting. We also rewrite all references within the function to point to the reallocated stack variables.

This transformation, however, requires adding a `smalloc` and `sfree` call to any function with sensitive stack variables. We found that these extra calls had a significant impact on performance (see Section 4.2), so we developed an alternative transformation for stack variables:

Shadow stack. A shadow stack is a separate area of memory that parallels the normal stack and holds sensitive variables. The shadow stack resides within the secure region, maintaining the invariant that all sensitive information is contained within that region. The shadow stack's size is set to the maximum size of the program's regular stack. We insert code to adjust the shadow stack pointer, which we implement as a global variable, at the entry and exit points of each function. This approach offers better performance than allocating all local variables on the secure heap.

Every time control reaches a function body entry point, the shadow stack pointer is incremented by the combined size of all of the sensitive variables for that frame. Thus, the shadow stack grows toward higher memory addresses.

We rewrite all accesses to variables declared with the `$sensitive` qualifier to use the new sensitive stack. We also insert code to decrement the stack pointer just before control leaves the end of the function body. After exiting a function, the memory located at higher addresses than the current shadow stack pointer is unused, but it still contains the remnants of the sensitive information that the function body placed there. We could overwrite the contents of this memory to eliminate the leftover values, but since the shadow stack is allocated within the sensitive region, it will be overwritten during the core file cleaning process anyway. Thus, overwriting the unused portion of the shadow stack is an unnecessary step, as the cleaning process will erase all of the stack contents, even the unused portions. See Section 3.5 for a description of the cleaning process.

3.4.3 Sensitive global variables

Finally, we define a new structure to contain all of the sensitive global variables, instantiating it as `__smalloc_global_var`. We allocate this structure on the secure heap with a special initialization function, using the `gcc`-specific attribute "constructor" to ensure that this function runs before `main()`. We also perform any initializations that are needed for sensitive global variables by expanding their initializer clauses into regular C statements and placing them in the constructor function.

3.5 Postprocessing: cleaning the core file

After employing the above transformations, all of the program's sensitive information will be fully contained within the secure memory region. The core file will still contain the sensitive information, however, if the program crashes and no further filtering steps are taken. At this point, we use a cleaning process to overwrite the secure region of a core file after it has been generated. The cleaner operates by first searching for a special tag that identifies the metadata for the secure region. The metadata encodes the type and size of the region, allowing the cleaning process to overwrite it.

Recall that the secure heap region is dynamically sized. When the region changes size, its new size is reflected in the metadata. If the region shrinks, memory that previously contained sensitive data will remain *outside* of the sensitive region. Thus, the cleaning process will not overwrite it. To maintain the invariant that sensitive data resides only within sensitive regions, Smalloc overwrites the contracted memory whenever a sensitive region shrinks.

The cleaning process must take special care to ensure that an adversary does not trick the cleaner into leaving portions of the sensitive region intact. Consider a cleaning process that scans through the core file sequentially, searching for the metadata that marks the boundary and size of a secure region and then erasing the specified number of bytes after the tag. A crafty adversary could arrange for a counterfeit secure region tag to appear in an insensitive memory region prior to the secure region, and construct the metadata so that the cleaner overwrites the actual secure region tag. Since the real secure region tag has at this point been erased from the core file, the cleaner would find no further tags and go on to generate a core file that still contains sensitive information.

To counter this attack, we wrote the cleaner to locate all secure region tags that might appear within a core file first, and then perform the overwriting. This approach prevents a metadata entry earlier in the core file from causing the cleaner to disregard a later one. Thus, all sensitive information will still be removed from the core file. An attack of this form may still induce the cleaner to remove insensitive data from the core file, but this is only a denial of service attack. The shortcoming doesn't represent a privacy or security threat, though it could hinder the developer from debugging the core file. We view guarding against denial of service attacks as a secondary concern, compared to protecting the user's privacy.

One could imagine incorporating core file cleaning into the operating system routines that produce the core file. Making this change would ensure that the cleaning process always runs before the crash report is written to disk, and would prevent problems such as the Dr. Watson bug mentioned in the introduction.

3.6 Implementation details

3.6.1 Threads

The Vmalloc package, on which the Smalloc allocator is built, is thread-safe, so extending our design to multi-threaded programs is straightforward. If sensitive local variables are transformed into heap-allocated structures, no changes to our system are necessary. Performance is a concern, however, since the many calls to `smalloc` in each thread will contend for the lock that guards the heap.

Alternatively, using the shadow stack approach to hold the sensitive local variables requires each thread to have its own shadow stack, just as each has its own traditional stack. The shadow stack pointer, which in single-threaded programs is simply a global variable, must therefore be stored into *thread-local storage*. Each thread has a pointer to its own shadow stack. The stack space for the thread is allocated during the first use of the shadow stack and freed when the thread terminates.

Since the same function may be called in different threads, each function with sensitive local variables retrieves the shadow stack pointer for the current thread upon entry to the function. When the pointer is updated, it must be stored into thread-local storage. For programs using POSIX threads, we add the following to the beginning of each function body:

```
void * stackPointer = pthread_getspecific(scrash_stack_key);
stackPointer += sizeof(struct function_shadow);
pthread_setspecific(scrash_stack_key, stackPointer);
```

and before each return statement, we add:

```
stackPointer -= sizeof(struct function_shadow);
pthread_setspecific(scrash_stack_key, stackPointer);
```

Note that the structure `function_shadow` holds the contents of all sensitive local variables for that function.

The first part of the structure's name identifies the function in which it is used.

An alternative to thread-local storage would be to add an extra parameter to every function that holds the address of the current thread's shadow stack. Unfortunately, it is often not possible to change the signature of every function, since functions such as event handlers and signal handlers are called by underlying systems.

3.6.2 setjmp / longjmp

A naïve implementation of the shadow stack will not correctly handle `setjmp` and `longjmp`. These functions are frequently used as a mechanism to pass control non-locally as an interprocedural goto, which is useful for error handling. The `setjmp` call saves the register contents, including stack pointer and program counter, in a `jmp_buf` structure. A `longjmp` call takes a previously populated `jmp_buf` as an argument and restores the registers saved in this structure. Since restoring the `jmp_buf` replaces the stack pointer and program counter, the stack is unwound and the program returns to the site of the `setjmp` call, this time with a non-zero return value from `setjmp`.

Scrash maintains its shadow stack by pushing a new frame upon entry to a function and popping it just prior to exiting the function. However, the default `longjmp` implementation is unaware of the Scrash shadow stack, and will not properly restore the shadow stack pointer as it does the regular stack pointer.

We address this problem by using CIL to introduce a new structure, `scrash_jmp_buf`, which replaces a regular `jmp_buf`. It has two fields: one to contain the old `jmp_buf` structure and one to store the shadow stack pointer. We then search for all calls to `setjmp` and `longjmp` and replace them with functions that properly maintain the shadow stack pointer in addition to the registers in `jmp_buf`.

Note that when calling `setjmp` in a threaded environment, we store the thread-specific shadow stack pointer (normally stored in thread-local storage) in the `jmp_buf`. This transformation is necessary because a thread's state in Scrash is described by the contents of the registers, stack pointer, and shadow stack pointer, all of which must be stored in `jmp_buf` for `longjmp` to work properly. On a `longjmp` call, we restore the stack pointer back into thread local storage.

3.6.3 Sensitive function arguments

The C calling convention places all arguments to a function on the call stack. Thus, calling a function with a sensitive value will place sensitive information on the unprotected call stack. Our solution to this problem does not require any effort on the part of the programmer; instead, a Scrash transformation converts a sensitive argument into a pointer reference to the sensitive data. Thus, the sensitive value is never placed on the call stack. Naturally, all such function bodies, declarations, and call sites need to be modified. To transform the call site, we first allocate space on the sensitive stack for any sensitive arguments. Then, we make a copy to preserve the call-by-value semantics of C and call the function with a pointer to the data.

Rewriting a function is not possible if the program exports a fixed API, passes a function pointer to a library callback function, or has a variable number of arguments. If Scrash detects that the address of a particular function is ever passed as an argument, it will refuse to modify that function, since changing its signature could yield unpredictable behavior. Instead, Scrash prints a warning advising the user of the security vulnerability. It is then up to the developer to modify the API to avoid passing sensitive variables by value.

4 Evaluation

We tested our system by applying the Scrash code transformations to a set of open-source applications and then comparing the behavior of each modified program to that of the original. We chose our set of test applications to include commonly-used graphical and command-line programs that handle significant amounts of user data.

Our first graphical test application was `gnomecal`, the calendar portion of the GNOME Personal Information Management suite. This application consists of about 25,000 lines of C code. Our other GUI-based test application was J-Pilot, a desktop organizer application for Palm OS-based handheld computers that contains about 42,000 lines of C code. It provides support for datebook, address storage, memo and "to-do list" handheld applications, while also facilitating PC-to-handheld data synchronization and backup. Both `gnomecal` and J-Pilot use the GTK+ graphical user interface libraries. When instrumenting these programs, we first examined the source code to determine which library I/O routines were most likely to be involved in the processing of sensitive user data. We then included ap-

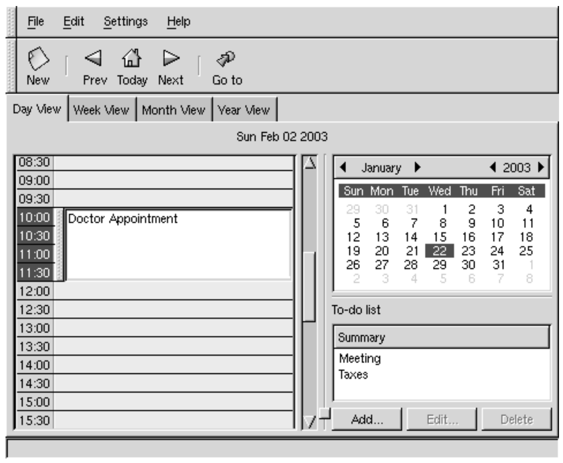

Figure 5: A screenshot of the GNOME Calendar application running with the Scrash transformations.

propriate declarations of these functions in a pre-annotated header file (as described in Section 3.2) prior to performing sensitivity inference on the program source code.

We chose the OpenSSH secure shell client, which contains about 39,000 lines of C code, as our command-line test program. For this application, it was necessary to treat all data typed by the user at the keyboard as sensitive. The password used to set up a secure connection is the most obvious sensitive value, but even after the connection is established, the client may send passwords and other private information to the server. Therefore, we again used pre-specified annotations to mark all data returned by `read` (among other functions) as sensitive.

After Scrash ran its compile-time type inference on our test applications, 24% and 10% of the stack variables used by `gnomecal` and J-Pilot, respectively, were marked as possibly containing sensitive data. For `ssh`, this figure was 59%.

We instrumented our Smalloc library to record the size and sensitivity of each run-time memory allocation request issued during the lifetime of a program. We then used each of the test applications for brief session. The run-time values from these tests are listed in Table 1. We only counted allocations performed by the application and not by any linked, precompiled libraries; this issue is discussed further in Section 5. The overall percentages of memory operations that dealt with sensitive data were lower for the graphical applications than for `ssh`. In `ssh`, the insensitive heap contains a few control structures representing the internal state of the connection, while the majority of the heap allocations are for sensitive user data that is to be transmitted over the network. We argue that the connection data is more relevant for debugging than the data being transmitted.

```
core.normal.dirty:

000732e0: 6d80 0608 7fd0 0708 0cf3 ffbf 0004 0000  m...............
000732f0: 0200 0000 34f7 ffbf e854 0908 98f8 ffbf  ....4....T......
00073300: 6842 0908 1800 0000 a066 2440 6162 7261  hB.......f$@abra
00073310: 6361 6461 6272 6100 5842 0908 f058 0908  cadabra.XB...X..
00073320: c830 0840 c4ef 0f40 7c3b 0908 28f4 ffbf  .0.@...@|;..(...
00073330: 28f4 ffbf 5842 0908 0000 0000 8855 0908  (...XB.......U..

core.smalloc.dirty:

0006a330: 70d0 2340 0000 0000 0000 0000 0000 0000  p.#@............
0006a340: 70d0 2340 0904 0000 0100 0000 0df0 edfe  p.#@............
0006a350: 6162 7261 6361 6461 6272 6100 0000 0000  abracadabra.....
0006a360: 80d3 2340 0000 0000 70d0 2340 0000 0000  ..#@....p.#@....
0006a370: 70d0 2340 0000 0000 90d3 2340 0000 0000  p.#@......#@....

core.smalloc.clean:

0006a330: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a340: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a350: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a360: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a370: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
```

Figure 6: Excerpts from the core file of an induced crash in the `ssh` client. The top core file excerpt shows the stack with the password present – “abracadabra” from an unmodified `ssh` client. The middle core file is from a version of `ssh` that has been modified using the Scrash transformations and annotations. The password now resides in the secure region, but since the cleaning process has not yet been executed on the core file, the password is again present. The bottom core file shows that the cleaner overwrites the secure region, and all occurrences of the password have been removed.

4.1 Security evaluation

We examined core files produced by our modified version of `ssh` to verify that sensitive information was placed only in the secure region and that the cleaning process properly eliminated sensitive data. Figure 6 shows the excerpts from three core files in which we induced a program crash. The top core file is the original version of `ssh`, in which the password is present on the stack. The middle core file is the result of running `ssh` after applying the Scrash transformations, in which the password resides in the secure heap. The final excerpt shows the result after running the cleaner.

4.2 Performance

Finding privacy-relevant and performance-critical applications proved to be a rather tricky exercise for us. Many of the applications for which one would be concerned about leaks of personal data were interactive: editors, browsers, information management tools, or remote access programs like `ssh`. In the course of testing, we found that the Scrash transformations did not reduce the responsiveness of the interactive applications we tested. In an attempt to better quantify the performance impact of Scrash, we ran two tests: one real application and a micro-benchmark to illustrate worst case behavior.

To test Scrash against a privacy-sensitive program that also has performance requirements, we chose to transfer

Size (bytes)	GNOME Calendar number of requests	GNOME Calendar percent sensitive	J-Pilot number of requests	J-Pilot percent sensitive	OpenSSH client number of requests	OpenSSH client percent sensitive
0 - 1023	4216	86.9%	5914	26.7%	2073	97.6%
1024 - 2047	9	77.5%	0	–	46	100%
2048 - 3071	1	0%	1	100%	67	100%
3072 - 4095	7	85.7%	1	100%	3	100%
4096 - 5119	2	50%	2	100%	50	100%
5120 - 6143	1	100%	0	–	3	100%
6144 - 7167	0	–	0	–	3	100%
7168 - 8191	7	100%	0	–	2	100%
8192 - 9215	0	–	0	–	1	100%
9216+	9	100%	0	–	12	100%
Total	4252	86.9%	5918	26.8%	2260	97.8%

Table 1: The number and size of all run-time memory allocations (`smalloc`, `scalloc`, `srealloc`) performed by our test applications during a brief test run and the percentage of these allocations that handled sensitive data. We only count allocations done by the applications and not by any libraries that they use.

Sensitive locals moved to:	**Elapsed Time(s)**	**Increase over baseline**
Heap	27.24	33%
Shadow stack	21.71	6%
Baseline	20.51	–

Table 2: Time needed for `scp` to transfer 100 megabytes of data to a server on the same machine. The results demonstrate that using a shadow stack gives much better performance than storing sensitive locals in the heap.

Sensitive locals moved to:	**Heap allocs**	**Elapsed Time(s)**	**Increase over baseline**
Heap	657393865	242.64	373%
Shadow stack	2	62.42	22%
Baseline	1	51.28	–

Table 3: Results of running the greatest common divisor (GCD) microbenchmark. We computed 50 million GCD computations on integers. The other two implementations place the sensitive local variables on the heap and shadow stack. The first column indicates the number of heap allocations that the microbenchmark makes. The results demonstrate that using a shadow stack gives much better performance than storing sensitive locals in the heap.

large files using `scp`. This program has many of the same privacy vulnerabilities as `ssh`, as it in fact calls the `ssh` executable. The program is non-interactive, allowing us to measure changes in performance easily. For the tests, we transferred a 100-megabyte file to a server on the same machine to eliminate network-induced performance variability.

The second test is a microbenchmark that exercises the call stack heavily. We wrote this recursion-intensive benchmark to expose the worst case performance of Scrash, since every function entry and exit requires intervention from Scrash. Furthermore, all locals are declared to be in the sensitive stack, increasing the normal memory access times. The microbenchmark computes the greatest common divisor of 50 million pairs of random numbers, where each number is between 1 and 10 million. The benchmark uses Euclid's algorithm, which admits a natural recursive implementation. Each invocation performs very little computation: a modulus call, two comparisons and assignments, and then a recursive call. Due to the heavy use of recursion, the amount of stack maintenance overhead that this microbenchmark incurs is significantly greater than that of a typical application. We used the same random seed when testing all implementations, processing 657,393,863 function calls per test run. To exercise the Scrash transformations, we marked all local variables as sensitive, as well as one global variable that we used to track the number of function calls.

We performed the above tests on a 1.5 GHz Pentium 4 with 1 gigabyte of RAM, running a Linux 2.4.18 kernel with gcc 2.95. All tests were run with optimizations turned on at -O3. The tests were conducted under three different configurations: without Scrash (the baseline), using Scrash to place all sensitive local variables on the heap, and finally using the shadow stack to hold the sensitive local variables. The results, shown in Tables 2 and 3, are based on the averages of three separate test runs per configuration. See Section 3.4.2 for a description of the two Scrash configurations.

Our initial strategy of moving sensitive stack variables to the heap via a call to `smalloc` at the beginning of each applicable function, as described in Section 3.4.2, resulted in a large performance penalty of 33% overhead for `ssh` and 373% for the microbenchmark. The microbenchmark suffers a larger overhead because it performs over 600 million allocation and free calls – one for every procedure entry. It also incurs a much higher percentage increase over the baseline because function entry time is a larger percentage of the CPU time for the microbenchmark than for `ssh`.

The second strategy, using Scrash transformations to implement a shadow stack, added much less overhead: 22% for the GCD microbenchmark and 6% overhead for `ssh` – a moderate overhead for the realistic application scenario. This overhead is a result of maintaining the shadow stack pointer at the beginning and end of the function, as well as the extra level of indirection required to access local variables.

We see that the shadow stack gives much better performance than placing sensitive locals on the heap. Consequently, we enable the shadow stack by default.

We conclude that Scrash adds only a minimal performance overhead to real applications.

5 Discussion

In addition to the runtime overhead imposed by Scrash, the system requires some effort from the programmer. This effort includes annotating an initial set of sensitive variables or deciding to use a pre-annotated "prelude" file that automatically marks the parameters and return values of certain functions as sensitive. In addition, it was necessary to make 33 lines of source code changes to `ssh` before it could run through the Scrash transformation, due to the fact that CIL is more restrictive in type checking than `gcc`. Such changes included fixing missing or mismatched variable declarations.

The performance of the Scrash code transformation tool is adequate. It takes roughly three minutes to run the entire Scrash transformation on `ssh`, from preprocessing through program modification, using the same test machine as above.

One feature of the `ssh` code was particularly problematic for Scrash: all calls to `malloc` are performed using a wrapper function, `xmalloc`, that checks for a null return value. Recall that Scrash rewrites calls to the `malloc` function to use `smalloc`, locating the new region on either the secure or insecure heap as appropriate. Since the `ssh` program calls the `xmalloc` wrapper, the only instance of `malloc` in the `ssh` source code is within the `xmalloc` wrapper. Scrash must choose whether to translate this `malloc` call into an allocation on the secure or insecure heap at compile time. Since the results of this allocation are assigned to some variables declared with the `$sensitive` keyword, Scrash conservatively translates the `malloc` call to allocate all its storage on the sensitive heap. As a result, all heap allocations in `ssh` would normally appear on the sensitive heap. To avoid this problem, we replaced the `xmalloc` function with an equivalent preprocessor macro at each allocation point. Thus, in the post-processed file, there is now one `malloc` call where each `xmalloc` call previously appeared, allowing the different `malloc` calls to be assigned to different heaps.

We must be a bit careful in evaluating the success of a technique like Scrash. For example, the absence of the password from the core file does not mean that there is no sensitive information related to the password in the core file. It may be possible to ascertain the size of a sensitive buffer by comparing pointers. If p is a pointer to a sensitive data field, an attacker can bound the size of the sensitive data by comparing all other heap-allocated pointers, t, to the sensitive data pointer:

$$\min_{t>p}(t-p)$$

That is, the size of the buffer at p is at most the difference between p and the first pointer whose value is greater than p. Thus, it may be possible to reveal the length of a variable-sized sensitive buffer even if all variables that explicitly store this length are kept in the sensitive memory region. This apparent vulnerability would seem to suggest that p is also sensitive and should be placed on the sensitive heap, adding an extra level of indirection to all accesses to p. We eschew this extra indirection, however, in favor of providing greater debugging usefulness to the developer, since hiding the pointer values may hamper the developer's ability to track down memory problems.

Another issue with Scrash involves the use of precompiled and dynamic (shared) libraries. Current libraries such as `glibc` are written without consideration of the concept of sensitive data. CQual understands the semantics of many `glibc` functions and will correctly propagate qualifiers across, for example, calls to `memcpy`. There is no way, however, for a source-level translation like Scrash to modify the storage of variables in precompiled libraries. For example, a precompiled version of `strcpy` may keep a char temporarily on the stack, or `strlen` may keep a running string length count as a stack variable. In the event of a crash, these variables will remain on the insecure stack, where they can leak pieces of sensitive information. One solution would be to recompile libraries with Scrash under the assumption that all data passed to a shared library is sensitive. The library would therefore use the shadow stack and sensitive heap so that sensitive data may be passed to the shared library without fear of privacy violations. However, we have not implemented this solution in our prototype.

As we discussed in Section 2.1, there are tradeoffs between user privacy and utility to the developer when dealing with crash information. Scrash provides the developer with a larger set of tradeoffs than the all-or-nothing choice that exists currently, while requiring minimal effort and time to specify and apply these tradeoffs to a program.

We believe that Scrash will help developers to allay users' privacy concerns about using crash reporting tools, and dissuade users from turning off the automatic crash reporting features in their applications. Widespread use of remote crash reporting will aid developers in improving the overall quality of software, in addition to helping make users aware of software patches for problems that they are experiencing.

6 Related Work

To the best of our knowledge, there has been no previous research published on the topic of limiting crash data to ensure privacy. The only other sources to mention this issue are the aforementioned Department of Energy advisory about Microsoft's Dr. Watson [12] and an online article on the same subject [18]. Both sources suggest that the user should disable crash reporting altogether to avoid a privacy risk.

Dr. Watson [4], the independent BugToaster application for Windows [2], the Bug-Buddy bug reporting tool for GNOME [1] and the Talkback quality reporting agent for Netscape/Mozilla [3] represent the current state of the art in remote crash reporting software. All are capable of sending back portions of the program's memory contents, including the registers, call stack and heap. Bug-Buddy is the least automated of the four, starting automatically when a GNOME program fails but then requiring a high degree of user participation to send a crash report. The other three require only the consent of the user via a dialog box to send a crash report.

The core file cleaning process is analogous to the scrubbing process that Gutmann advocates for securely deleting sensitive information from media, such as RAM or magnetic media [10]. His cleaning process is aimed at protecting against physical attacks against storage media that are not easily erasable. Other work focuses on creating a large block of erasable memory from a much smaller block using cryptographic techniques to achieve similar ends [6]. In contrast, we view our cleaner as operating on the *contents* of files to eliminate sensitive information so that they may be safely sent over the network.

There is a large body of work that describes techniques for efficient allocators [17] and garbage collectors [16]. Region-based memory allocators in which multiple heaps are exposed have also been studied [7, 8]. While they present a richer set of semantics than we need, these sources helped to inspire our implementation. We used the Vmalloc software release as the basis for Smalloc, our secure memory allocator [15]. Vmalloc provides an alternative allocator to `malloc` that exposes many different allocation fit strategies and provides rich internal interfaces.

We use CQual, a static analysis tool, to track the possible spread of sensitive information [14]. Sabelfeld and Myers [13] survey language-based systems for statically tracking information flow in a secure manner. Tracking information flow typically involves removing covert channels within a program, which can require extensive code modifications. While information-flow concerns are a central theme of this work, we do not address the issue of convert channels.

7 Future Work

Changes to Scrash in the short term mostly involve improvements to the analysis phase. The implementation of CQual that our current system uses is at times too conservative – it marks too many variables as `$sensitive` – but we expect to be able to use a more accurate version soon. The new implementation, currently under development, will use a polymorphic analysis of functions so that more variables can be safely labeled insensitive. Modifying Scrash to work with C++ is another area of active interest; CQual has recently been extended to work with C++ code.

In addition, we hope that support for Scrash will be incorporated into some of the standard bug reporting tools, such as the GNOME Bug-Buddy. Another avenue would be to combine runtime error detection tools, such as StackGuard or CCured [5, 9], with Scrash. When these runtime tools would detect a violation, Scrash would send a core file to the developer. This pairing would aid in the detection of security vulnerabilities such as buffer overruns.

8 Acknowledgments

Many people have contributed to this project. Dan Wilkerson and Rob Johnson implemented many last-minute CQual features for us, while John Kodumal, Jeff Foster, and the rest of the CQual team provided advice on using CQual. We thank Ben Liblit, David Gay, and Jeremy Condit for their insightful comments and suggestions. Finally, David Wagner provided helpful guidance along the way.

References

[1] Jacob Berkman. Project Info for Bug-Buddy. `http://www.advogato.org/proj/bug-buddy/`,

2002.

[2] Bugtoaster. Do Something about Computer Crashes. `http://www.bugtoaster.com`, 2002.

[3] Netscape Communications Corp. Netscape Quality Feedback System. `http://wp.netscape.com/communicator/navigator/v4.5/qfs1.html`, 2002.

[4] Microsoft Corporation. Dr. Watson Overview. `http://www.microsoft.com/TechNet/prodtechnol/winxppro/proddocs/drwatson%_overview.asp`, 2002.

[5] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[6] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to Forget a Secret. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, number 1563 in Lecture Notes In Computer Science, 1999.

[7] David Gay and Alexander Aiken. Memory Management with Explicit Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[8] David Gay and Alexander Aiken. Language Support for Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[9] Scott McPeak George C. Necula and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Principles of Programming Languages*, 2002.

[10] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Sixth USENIX Security Symposium Proceedings*, 1996.

[11] George C. Necula, Scott McPeak, Westley Weimer, Raymond To, and Aman Bhargava. CIL: Infrastructure for C Program Analysis and Transformation. `http://www.cs.berkeley.edu/~necula/cil`, 2002.

[12] U.S. Department of Energy Computer Incident Advisory Capability. Office XP Error Reporting May Send Sensitive Documents to Microsoft. `http://www.ciac.org/ciac/bulletins/m-005.shtml`, October 2001.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information Flow Security. *IEEE Journal on Selected Areas in Communications*, January 2003.

[14] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium*, pages 201–220, August 2001.

[15] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.

[16] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[17] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[18] Brandon Wirtz. Dr. Watson's a Big-Mouth. `http://www.griffin-digital.com/dr__watson.htm`, 2002.