2017 IEEE 41st Annual Computer Software and Applications Conference

SecretSafe: A Lightweight Approach against Heap Buffer Over-read Attack

Xiaoguang Wang*, Yong Qi*, Chi Zhang*, Saiyu Qi† and Peijian Wang*
*Xi'an Jiaotong University, China
†Xidian University, China
Email: xiaoguang@stu.xjtu.edu.cn, qiy@xjtu.edu.cn

***Abstract*—Software memory disclosure attacks, such as buffer over-read, often work quietly and would cause secret data leakage. The well-known OpenSSL Heartbleed vulnerability leaked out millions of servers' private keys, which caused most of the Internet services insecure at that time. Existing solutions are either hard to apply to large code bases (e.g., through formal verification [20] or symbolic execution [8] on program code), or too heavyweight (e.g., by involving a hypervisor software [23], [24] or a modified operating system kernel [17]).**

In this paper, we propose SecretSafe, a lightweight and easy-to-use system which leverages the traditional x86 segmentation mechanism to isolate the application secrets from the remaining data. Software developers could prevent the secrets from being leaked out by simply declaring the secret variables with `SECURE` keyword. Our customized compiler will automatically separate the secrets from the remaining non-secret data with an isolated memory segment. Any legal instructions that have to access the secrets will be automatically instrumented to enable accesses to the isolated segment. We have implemented a SecretSafe prototype with the open source LLVM compiler framework. The evaluation shows that SecretSafe is both secure and efficient.

***Index Terms*—Buffer over-read attack, vulnerability elimination, memory segmentation.**

1. Introduction

Secret data, such as private keys, user identities or financial information, are often maintained in and by the running server program. A simple code vulnerability such as the buffer overflow, the out-of-bound memory read, or the off-by-one error [38], could make those information leaked out easily, which would cause tremendous sensitive data leakage or large scale service insecure. A motivating example would be the well-known OpenSSL vulnerability, Heartbleed (CVE-2014-0160) [14], with which an attacker could use the vulnerable OpenSSL heartbeat code to read the whole heap memory and further dump out the private server keys. This caused millions of data encrypted with SSL protocol under the hood of the attackers.

A direct way to solve these problems is to use type safe languages, such as Java or C#, to rewrite the entire server software stack. Needless to say the huge engineering efforts, those type safe languages are often considered as slow and inefficient [37]. Another way to solve the above-mentioned problem might be using the formal verification [20] or symbolic execution testing [8] to verify the source code against potential memory vulnerabilities. However, such code verification techniques are often suffered from execution path explosion problems, and thus are very hard to scale for a larger code base [6], [8], [20] (e.g., OpenSSL contains over 400,000 lines of C code in the latest stable version). Very recently, researchers have proposed to use an underlying hypervisor [23], [24] or a modified operating system kernel [17] to build the isolated compartments for those sensitive data. Although they work appropriately, it is far too bloated to address a user space security problem by involving a hypervisor or a specialized kernel. In fact, due to some compatibility issues, it might be difficult for some existing cloud services to completely upgrade the underlying privileged software [26].

The secret data leakage problem is fundamentally caused by that the secret data and the other parts of the program share the same address space. Therefore, an over-read of the memory buffer would possibly reach the application secrets. The existing programming languages and the commodity processors are not designed with the secret data separated in concern. Existing security enhancing systems, such as stack guard [11], CFI [1], DFI [9], or CPI [21], could not address this over-read problem neither. This is because the approaches above are all focus on preventing the data, e.g. the code pointers, from being modified. On the other hand, a buffer over-read could happen stealthily. For example, a buffer over-read will not cause a guard value, such as a canary [11], being changed.

In this paper, we propose SecretSafe, a lightweight and easy-to-use approach to protect the secret heap data from being leaked out in a running application. SecretSafe allows the application programmers to mark the secret data by simply adding a `SECURE` keyword in front of the variable definition (as shown in code listing 1). A modified SecretSafe compiler further transforms and generates the machine code with secret data protected. With SecretSafe, only legal secret data accesses will be allowed while any illegal data accesses, such as a buffer over-read [33], will be denied by the hardware memory isolation.

SecretSafe leverages the long-existing segmentation-

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.206

based isolation mechanism in x86 platforms to separate secret data out of the existing process address space. Segmentation has been proposed long ago to support the early x86 virtual memory system, but it was later superseded by paging [18]. For backward compatibility reasons, current operating systems set each of the segment's boundary as the whole virtual address space. For example, the code and data segments (confined by the segment selectors pointed by `CS` and `DS` registers respectively) in 32 bit processor mode are all set with address range of 0∼4GB [7]. In SecretSafe, we reserve a small range of virtual address space for the secure segment to store the secrets, and use a reserved segment register `FS` to point to that secure segment. SecretSafe further re-configures the `CS` and `DS` registers to exclude the secure segment out of their accessing boundaries. Therefore, any (potentially illegal) regular application code would not be able to access the secrets inside the secure segment, because the processor by default will access the data confined by the `DS` segment. While for those legal secrets accesses, our customized compiler tracks and analyses the variables declared with `SECURE` keyword, and further generates the secrets accessing machine code (those code pieces have explicit `%fs:` prefix in the secret data access instructions).

SecretSafe provides two main features to protect secret data in a program: (1) ease of use: what application programmers need to do is just registering the application secrets with SecretSafe helper code (a few lines of code modification) and compiling the modified application code with the SecretSafe compiler; (2) low runtime overhead: most of the works are done during the compiling and executable loading phases, which eliminates most of the runtime overhead. We have built a prototype of SecretSafe with the open-source LLVM compiler framework [22].

The rest of this paper is organized as follows: Section 2 gives a motivating example of the problem. We then describe the design and the implementation of SecretSafe in Section 3. The evaluation is presented in Section 4. We list the related work in Section 5 and summarize the paper in Section 6.

2. Problem Overview

Code listing 1 shows an example of adding the `SECURE` keyword for the potential secrets in RSA data structure in OpenSSL. The RSA encryption algorithm implements an asymmetric encryption with two large random prime numbers `p` and `q`. By using some mathematical (e.g. modulo) calculations, it calculates the public key `<n,e>` and the private key `<n,d>` (where `n` equals `p * q`). For the calculation speed consideration, OpenSSL maintains all these intermediate values in the RSA struct, including `p`, `q`, `d` and `e`. Code listing 1 shows some of these variables in `struct rsa_st`. Since `<n,e>` is publicly known, the attacker could retrieve the private key (a.k.a `<n,d>`) by obtaining any of the numbers `d`, `p` or `q` [1]. Therefore, we should mark the variables `d`, `p` and `q` as `SECURE` variables. Since the real secret data is pointed by `BN_ULONG *d` in `struct bignum_st` (i.e. `BIGNUM`), we should also mark `BN_ULONG *d` as `SECURE` variable. After that, the memory objects pointed by the `SECURE` variables are considered as application secrets, which will further be relocated into a separated memory region and protected by SecretSafe.

However, tagging the `SECURE` variables' declaration, such as `d`, `p` and `q`, alone could not totally solve the problem. The secret memory objects could be referred by other program variables through, for example, pointer assignments. Pointer assignment could possibly make a non-secret variable point to a secret object. This will potentially lead to a larger `SECURE` variable set. We define this variable set as the *potential* `SECURE` *variable set*, and the variables in this set could possibly point to the secret objects (pointed by a `SECURE` variable). For example, in code listing 1, the parameter `const BIGNUM *a` of function `BN_get_word` could possibly point to a secret at runtime. Therefore, the actual generated code that access variable `a` will have two situations, one for accessing a secret object `a` and one for a non-secret `a`.

Listing 1: OpenSSL RSA private key protection

```
/* crypto/bn/bn.h */
typedef struct bignum_st
 {
 SECURE BN_ULONG *d;   // The real bytes array
  int top;
  int dmax;            // Size of the array
        ... ...
 } BIGNUM;

/* crypto/rsa/rsa.h */
struct rsa_st
  {
        ... ...
  BIGNUM *n;
  BIGNUM *e;
  SECURE BIGNUM *d;
  SECURE BIGNUM *p;
  SECURE BIGNUM *q;
  BIGNUM *dmp1;
    ... ...
  };
          ... ...
/* crypto/bn/bn_lib.c */
BN_ULONG BN_get_word(const BIGNUM *a)
  {
  if (a->top > 1)
          return BN_MASK2;
  else if (a->top == 1)
          return a->d[0];
  /* a->top == 0 */
  return 0;
  }
```

Threat model and assumption: we assume the attackers could exploit any program vulnerabilities to over-read the memory they need. Such program vulnerabilities could be the buffer overflow, the out-of-bound memory read or the

1. Attackers could easily get `p` by using `n` divides `q`, and the other part of the private key `d` could be calculated by using the modulo calculation of `p` and `q`.

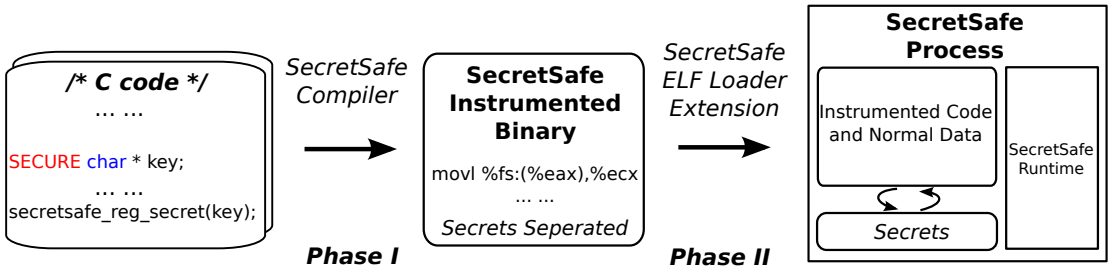

Figure 1: SecretSafe Overview

off-by-one errors, which just give the attackers extra memory information. Once the attackers get the extra memory content, they could retrieve the important user information, such as the private keys or the user identification [14]. Such attack model is reliable and recent surveys show that there are large possibility for attackers to obtain secrets in a running process [23], [33]. In our threat model, we also assume the secrets are allocated and stored on heap. This is because the process stack can only temporarily hold the data (stack data are released when functions return), thus it is not suitable to keep secrets on stack. We do not aim at protecting the program from other memory corruption attacks, such as a pointer hijacking [31] or an integer overflow [13]. We believe that the existing approaches [1], [9], [21], [32], [34] could have addressed those problems. Another assumption is that the operating system and even the underlying hypervisor are secure. We believe the existing techniques, such as secure kernels [4], [12], [20], [35] or trusted hypervisors [24], [36], could provide a secure process execution environment. And those existing defence techniques could work in orthogonal with the security guarantee from SecretSafe.

3. System Design

3.1. Overview

SecretSafe aims at automatically separating and isolating the secret data from the original code. To achieve that, SecretSafe first allows the programmers to identify the secrets by simply adding a `SECURE` keyword in front of the secret variable declaration. Since a secret variable could be defined at any place of the code (e.g., in a function body, or even inside of a structure) and could be referred by any C/C++ variable types, we define the `SECURE` keyword as a type qualifier [2]. A modified compiler further analyses the annotated source code and produces the instrumented binary (Phase I in Figure 1). The binary has all the code that would access secret variables instrumented, and those instructions that access the secrets will have a dedicated segment register prefix (we call those instructions the *secure instructions*, as compared to the original *normal instructions*). The secrets are stored in an isolated segment memory at runtime, so that any normal instructions without explicitly switching the segments cannot access the secrets. The idea seems straightforward and easy to be implemented. However, the C/C++ allows a pointer to dynamically refer to any values, and this property makes the SecretSafe design complex.

In order to address this problem, SecretSafe compiler first analyses the code, finds out all the secret variable references and instruments those instructions that access the secret variables. For those direct variable references, SecretSafe would create the code snippets for accessing secrets inside the isolated segment. Specifically, SecretSafe creates the data accessing instructions with a segment register prefix (e.g., `movl %fs:(%eax),%ecx`). The application secrets are stored in the isolated memory region specified by the explicit segment register (e.g., `%fs`), while the `%eax` stores the offset of that variable to the beginning of the secure segment. Any other memory accessing instructions without being instrumented cannot access the secure segment. The x86 platform has supported the segmentation-based memory isolation long ago, while it slightly changed the segment specification on recent 64-bit processors (a.k.a. x86_64). The 32-bit x86 processor allows the system programmer to set up a segment with a base address and a segment length, while the x86_64 processor only allows to set up a segment with the base address, leaving the segment length unlimited. This slight difference makes the SecretSafe code generation different on each platform.

On 32-bit x86 platforms, SecretSafe instruments all the possible instructions to access secret variables with a `%fs` prefix. The SecretSafe runtime narrows down the default `DS` segment, excluding the topmost memory region for the secure memory segment (as show in Figure 2 (a)). Therefore, any buffer over-read attempt to access the secret memory will trigger a CPU general-protection error. On the other hand, `FS` is configured as the whole memory space, making those secret accessing instructions correct as usual. *On x86_64 platforms*, we can no longer set the segment length (as we narrow down the `DS` segment limit on 32 bit platform). Instead, we put the secrets at a random place in the memory space (as show in Figure 2 (b)). Attackers could hardly guess out the secret location because of the large address range and the high entropy in the 64-bit address space.

The secrets are registered with a SecretSafe library function `secretsafe_reg_secret(void **ptr, uint_t size)`. This function allocates the isolated memory for the secret, moves the secret value into that

2. The type qualifier in C/C++ or D programming language is a keyword that is applied to a type. Other commonly seen type qualifiers include `const` and `volatile`.

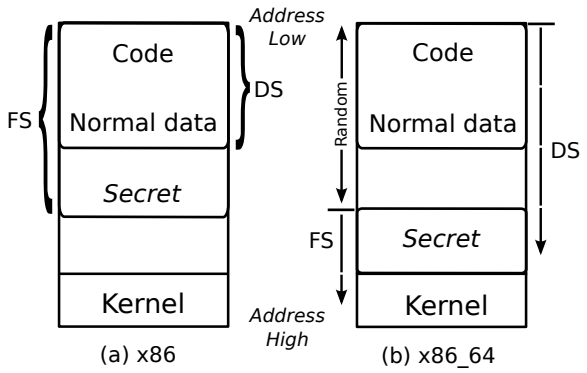

Figure 2: Segments in SecretSafe on x86 and x86_64 platforms

memory region and erases the original secret memory content. The secret is then stored in the isolated memory region enforced by the hardware segmentation. On 32-bit x86 platforms, secret data are moved to the topmost of the user space. The re-configured data segment prevents buffer over-read to the secret memory (Figure 2 (a)). While all the secret accessing instructions (including all the possible secret variable references) are instrumented with a `%fs` prefix, making them able to access both normal data and secret data.

On x86_64 platforms, however, the value of a secret variable will be assigned to a small unsigned value which is an offset to the beginning of the isolated segment (specified by `FS`). The isolated memory segment will be allocated at a random place by the SecretSafe runtime (Phase II in Figure 1), making attackers hard to locate the secrets. There could be some secret variable references that may interleavingly access the secrets or the non-secret data. For example, a pointer assignment could assign a `SECURE` variable to a non-`SECURE` variable. In this case, SecretSafe generates two data accessing paths, one for each case. While for the normal data access, SecretSafe just emits the original instructions (without the `%fs` prefix). The value of that variable at runtime is used to select the correct data accessing path (e.g. through a conditional branch instruction).

Programming effort with SecretSafe: The programming effort to adopt SecretSafe is quite small. As illustrated in Figure 3, it is an option for the application programmers to add the `SECURE` keyword in front of the variables which host the keys. The programmers would then involve a registration routine right after the secret variable has been initialized. The external registration routine will move the secret to the isolated memory region and erase the original secret memory. In the following of this section, we will describe in detail how each part of SecretSafe works together to protect the user defined secrets.

3.2. Secrets Separation and Isolation

Secret variables annotated with `SECURE` keyword will be handled by the modified compiler. The compiler analyses all the code snippets that access the secrets and further

```c
#include <stdio.h>
#include <stdlib.h>
#include <secretsafe.h>
#define SECURE __attribute__((type_annotate("secure")))

int main(int argc, char* argv[], char* env[])
{
    SECURE int *key;
    int *ptr;

    key = (int *) malloc(16*sizeof(int));
    secretsafe_reg_secret(&key, sizeof(key));
    *key = 0xa001;
    printf("key: 0x%x\n", *key);

    ptr = key;
    *ptr = 0xa003;
    printf("ptr: 0x%x\n", *ptr);

    return 0;
}
```

Figure 3: Programming effort with SecretSafe

instruments the secrets accessing instructions. SecretSafe protects the secrets on heap (e.g. memory allocated by `malloc()` etc.). However, SecretSafe does not directly monitor the heap allocation functions, such as `malloc()`. Instead, it moves the secrets into the isolated memory right after the secret memory has been allocated (with the help of a registration routine). SecretSafe is essentially a segmentation-based isolation approach [21], [39]. It leverages an unused LDT (Local Descriptor Table) entry to set up an isolated memory region for the secrets. Moreover, SecretSafe further modifies the existing code and data segments to exclude the reserved memory. Therefore, any user-space instructions could not access the isolated memory region except for explicitly updating the segment registers.

In order to help the application programmers hiding the secrets, SecretSafe provides two simple yet efficient library functions:

Listing 2: SecretSafe library functions

```c
/* SecretSafe.h: library functions for
    secret registration */
// Register a secret region pointed by ptr
    with the lenghth size
void secretsafe_reg_secret(void **ptr,
    uint_t size);
// Verify whether a region pointed by ptr
    has been registered as secret
int secretsafe_has_registered(void *ptr);
```

The first function registers the secret with a given address and length, while the second one helps to verify whether an address has been registered as a secret value. The register library functions as well as the `SECURE` keyword together help programmers to separate the secure variables out of the program space.

Code Instrumentation: Having the secret variables explicitly tagged, the compiler instruments the legal code snippets to correctly and safely access the secrets. Therefore, it is necessary to know which variable would possibly keep the secret data at runtime. As we have mentioned, secret

variables include those pointers declared with the SECURE keyword. We refer to those variables as the *initial secret variable set* $\{initV_s\}$. Pointer assignment could potentially pass the secret variables (declared by the SECURE keyword) to the variables without SECURE attribute. As such, the SECURE attribute may spread across the whole variable set. Similarly, a secret variable could be de-annotated the SECURE attribute by being assigned with a non-secret variable. Thus it is actually hard to know the accurate secret variable set due to the dynamic value assignment feature of a pointer. Obtaining the precise secret variable set is essential a data flow analysis problem [3]. One way to address this problem is to use the dynamic taint analyzing and tracking techniques [28], such as DDFT [15], [19]. However, one inadequate fact of DDFT is that it incurs large runtime performance overhead. SecretSafe instead tracks and tags all the variables that has the value assignment. During code generation phase, SecretSafe collects all the possible secrets by analyzing the intermediate code, and deduces the *possible secret variable set* $\{possiV_s\}$. The $\{possiV_s\}$ are the actual program variables that need to be instrumented for secret data accessing. Here is the pseudo code for calculating the possible secret variable set:

Algorithm 1 Possible Secret Variable Set Calculation

Initialize: $\{possiV_s\} \leftarrow \{initV_s\}$
repeat
  $\forall V_k \in \{possiV_s\}$
  if $\exists V_t \leftarrow V_k$ **then**
    set $V_t \in \{possiV_s\}$
  end if
until $\{possiV_s\}$ does not change

Once having collected the possible secret variable set $\{possiV_s\}$, SecretSafe generates the secret accessing code for each variable appearance in $\{possiV_s\}$. As we have mentioned, a possible secret variable at runtime could possibly point to a secret value or a non-secret value. The SecretSafe 32-bit x86 segmentation allows the possible secret variable accessing instructions unmodified to access normal data (Figure 2). For x86_64 platforms, SecretSafe should generate the code that could facilitate accessing both secret and non-secret values during runtime. To this end, SecretSafe creates two data accessing paths, one for each case. When holding a normal data address, the variable points to the data sections which are usually located at a higher virtual address [3]. While holding a secret, the possible secret variable stores the offset to the beginning of a segment. Thus it should be a small unsigned integer that is often less than a few pages' size, depending on the size of the secret. As such, we could use this fact to distinguish whether a variable stores a secret or non-secret at runtime. Code listing 3 shows the generated assembly code for accessing a possible secret variable ptr (`*ptr = 0xa003;` in Figure 3).

3. On i386 Linux, the program code and data are usually located at addresses above `0x8048000`; on x86-64 Linux, that lowest address bound by default is `0x400000`.

Listing 3: Possible secret variable accessing paths

```
1   jmp  .LBB0_3
2 # BB#1:
3   movl    $8192, %eax
4   movl    -40(%rbp), %ecx # -40(%rbp): ptr
5   cmpl    %eax, %ecx
6   jae .LBB0_3
7 # BB#2:                      # secret path
8   movl    -40(%rbp), %eax
9   movl    $40963, %fs:(%eax)
10  jmp .LBB0_4
11 .LBB0_3:                     # non-secret path
12  movl    -40(%rbp), %eax
13  movl    $40963, (%eax)
14 .LBB0_4:
```

`BB#1` and `BB#2` are two basic blocks instrumented by SecretSafe, and they are instrumented just before the actual data accessing instructions (Line 12, 13). Instructions in `BB#1` load the `ptr` value and compare the `ptr` against the segment boundary (usually several pages' size, e.g., 8196 in this case, Line 3). If the `ptr` value is lower than the boundary, SecretSafe could deduce the `ptr` has the SECURE attribute and it will instrument the data accessing instruction with a segment register prefix (Line 9). Otherwise, `ptr` will be treated as the normal pointer with the original data accessing path (go through `.LBB0_3`). Note that the first instruction in code listing 3 is a direct jump to `.LBB0_3`, which skips all the instrumented instructions of the secret data accessing path. This will cause the data accessing instruction a segmentation fault when `ptr` stores a SECURE value (i.e., an offset to the beginning of an isolated segment). SecretSafe handles the fault with a lightweight code monitor (described in the next part). The monitor captures the fault at runtime, verifies the fault reason and patches the code (by updating the destination address of the direct jump). By this way, SecretSafe reduces the performance overhead for accessing those possible secret variables, when they never host a secret.

SecretSafe Runtime: SecretSafe selects out all the variables that potentially contain the secrets' addresses in the first phase. It further instruments all the instructions that access those variables. The possible secret variable may have secret value or non-secret value, and it is determined at runtime. SecretSafe deals with the undetermined secret variables by instrumenting two data access paths for each of their appearance. However, the over-instrumented code could potentially lead to larger performance overhead, especially when there is a quite large possible secret variable set. Instrumenting the variables that are never assigned to a secret value at runtime could have unnecessary performance loss. To minimize the introduced performance overhead, SecretSafe skips the instrumentation that will never be used at runtime, while still keeps a minimal set of necessary instrumentation.

As we have shown in code listing 3, the instrumented code by default will be skipped by a direct `jmp` instruction. Therefore, no actual instrumentation code will be involved when the program starts. SecretSafe employs a runtime code

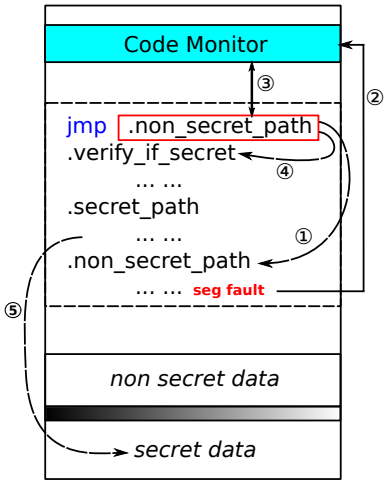

Figure 4: Code monitor that patches the secret accessing code

monitor to patch the code, and it turns on each of the patched code whenever the corresponding variable contains the secret. Figure 4 shows how the SecretSafe code monitor works to dynamically enable the instrumentation code. All data access on the variables that possibly hold the secret (address) will be instrumented with a code block with two data accessing paths. The first instruction skips the code block with a direct jump to the non-secret path (① in Figure 4). This helps to avoid executing unnecessary instructions when the variable never access the secret. However, for the variables that do have accesses to the secret, it will obviously cause program errors on non-secret path (② in Figure 4). This is because when accessing the secrets, the variable contains an offset (from the beginning of the isolated memory region) instead of a real virtual address. The offset is usually less than a few pages' length, thus without instrumentation, code instruction will access the virtual address an offset value from address *zero*. A segmentation fault occurs on such instruction.

The inline code monitor intercepts the segmentation fault, and verifies the fault reason. For the segmentation fault caused by secret data accessing, the code monitor patches the instrumentation (③ in Figure 4). Specifically, it updates the destination address of the leading `jmp` instruction, and changes it to the beginning address of the secret path. The code monitor then updates the IP register and gives control back to execute the secret path (④ in Figure 4). After that, the instrumentation for the secret access will be turned on, and it will be involved automatically the next time code access the secret data (⑤ in Figure 4). With the code monitor, SecretSafe could avoid the additional performance overhead caused by those possible secret variables that never access the secrets at runtime.

3.3. A Prototype Implementation

We have built a prototype of SecretSafe code instrumentation on the open-source, modular LLVM compiler framework. Many programming languages, such as C/C++, D, Go, and FORTRAN, have a LLVM front-end that generate LLVM IR (Intermediate Representation). Several passes could be implemented to analyse and translate the IR into the final code. We used the LLVM Quala project [2] to enable adding the `SECURE` keyword as a type qualifier. The annotated source code is further analysed by an analysis pass to deduce the possible secret variable set (Section 3.2). Code instrumentation is implemented in a separate translation pass, which instruments the IR and generates the code to access secure variables. All the implementation follows the SecretSafe design described in Section 3.2.

To clear out the secret memory contents, application programmers (or security analysts) are also required to register the secret variables after they have been initialized (e.g. after the `malloc()` function). The library functions mentioned in Section 3.2 help to register an allocated secret. The registration routine moves the secret data to the reserved memory region, and clears the secrets' memory contents. This facilitates the secrets to be protected by the hardware segmentation-based isolation.

We have implemented the runtime code monitor as a shared library, namely *libmonitor.so*. We use the Linux *LD_PRELOAD* mechanism to pre-load the shared library into the process address space (before the `main()` function starts). The monitor code prepares the isolated memory by setting up the segmentation as well as the segment register (e.g., `fs`). In SecretSafe prototype, we filled an unused LDT entry with the secure segment descriptor. We also modified the existing segment descriptors to exclude secure segment out of the existing segments' memory access ranges. The code monitor library also implements a user space segmentation fault handler (with `sigaction()` system call) to intercept the fault instructions. Since the segmentation faults are most likely caused by accessing a small offset from the virtual address *zero* [4], the monitor could deduce the instruction is accessing a secret variable. The fault handler disassembles the code region above the fault address, and patches destination address of the direct `jmp` instruction to the beginning address of the secret path [5]. After that, the secret path is enabled and any data accesses afterwards will be verified to use the correct data path.

4. Evaluation

In this section, we first evaluate the security improvements for the applications by SecretSafe, and then report the SecretSafe performance impact on the protected program.

4.1. Security Evaluation

SecretSafe is designed to protect secret heap data from buffer over-read attack. We assume the (remote) attackers

4. It supposed to be the offset from the beginning of a segment. The first several pages in the virtual address are often marked as NONE.

5. The code pages are usually set as read-only, thus the code monitor would temporarily set them writeable by using `mprotect` system call.

could use the existing program vulnerabilities to trigger a buffer over-read, which leaks out the confidential data, e.g., a private key or password. In order to get the secret, attackers might have to use the following two methods. First, attackers might try to retrieve the secret memory by over-reading the memory buffer. SecretSafe prevents this attack by separating the secrets from the program address space. The secrets are isolated into a reserved memory region, thus any memory accesses crossing the memory boundary will be prevented by the hardware-based isolation mechanism, i.e., the segmentation. SecretSafe will terminate the process on detecting the out of segment bound data access.

Second, attackers might try to get the secret value out of the heap or by pointer dereference. This can be prevented by SecretSafe. SecretSafe hooks the memory copy related functions and checks whether the source address has already been registered as secret. If the source belongs to the possible secret set, SecretSafe will also register the destination address. SecretSafe prevents the secrets from being referred by a non-secret variable. Such a case would cause a segmentation fault and will be further captured by the code monitor. The code monitor verifies the segmentation fault caused by the non-secret variable access, and it will raise an alert if there is no corresponding instrumentation code before the fault address.

To further test the SecretSafe's effectiveness on protecting secret data, we use the *heartleach* tool [6] to dump the private keys in a vulnerable OpenSSL server (version 1.0.1f). *Heartleach* is an open source OpenSSL heartbleed exploitation tool, which recursively dump the remote server memory until it gets the private key. In our test, we use the OpenSSL `s_server` to listen for the incoming SSL connections. Without SecretSafe's protection, *heartleach* can retrieve the private key in one second. While with SecretSafe's protection, *heartleach* failed to find the private server key. We also wrote a small memory dump program to dynamically attach to the `s_server` process (with `ptrace`) and search all the mapped address space. The results showed that SecretSafe can completely protect the private keys from being found out.

4.2. Performance Evaluation

To evaluate the performance impact of SecretSafe, we did two micro benchmark tests, measuring the pure value assignment costs and the network ping-pong cost, and two application benchmarks with Nginx web server and MiniZip. All the experiments are conducted on a Ubuntu 14.04 server (kernel version 4.1.27) with 2.5GHz Intel Xeon E5649 CPU and 32GB memory. For the experiments that require the network connections, we used a Dell XPS desktop as the client. The client machine has an Intel Core i7 CPU and 8GB memory. We measured SecretSafe's performance on both x86 and x86_64 platform respectively.

The first micro benchmark measures the cost of accessing a secret variable protected by SecretSafe and a non-secret variable (the baseline) one million times in a for-loop, assigning a random number to that variable. On x86 platform, SecretSafe incurs no observable performance overhead on the secret accesses. This is because SecretSafe on x86 uses the segment limit to protect secrets from being over-read. It only relocates the secrets to the topmost address of the process, changing very few in memory access patterns. While on x86_64 platforms, SecretSafe has to instrument the dual data accessing paths for the secret value, thus it brings several additional instructions (e.g., `cmpl`, `movl` etc., as we have shown in Code Listing 3). Those additional instructions actually occupy a relatively large portion of the instruction cycles for a simple for-loop code like our benchmark. Thus it is reasonable for SecretSafe on x86_64 to have a worse result (`micro-1` in Figure 5). The second micro benchmark measures a network ping-pong test, in which a client sends an unsigned integer value (protected by SecretSafe) to the server and the server returns back the value plus one back to client. The procedure above was repeated 10,000 times and the total time cost was measured. We can see that SecretSafe only brings about 6% performance degradation compared with unprotected benchmark (`micro-2` in Figure 5).

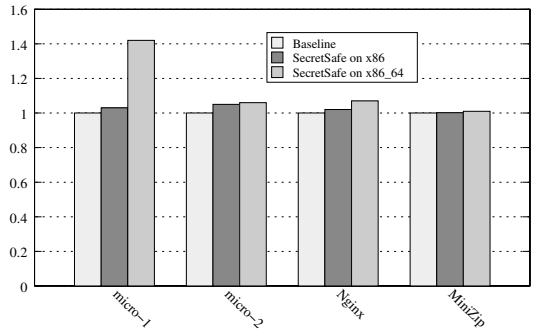

Figure 5: Performance degradation on applications with secrets protected by SecretSafe. Execution time is normalized to the baseline test

We also measured the SecretSafe's impact on real applications. In the first test, we compiled the Nginx web server with SecretSafe protected OpenSSL library. We used the ApacheBench (ab) to test the throughput of the protected Nginx/OpenSSL and the unprotected Nginx/OpenSSL (the baseline). SecretSafe brings about 2% and 7% of performance degradation on x86 and x86_64 respectively (`Nginx` in Figure 5). The SecretSafe compiled code has more CPU resource occupation during the SSL tunnel establishment (a.k.a., key creation and exchange), which involves a lot of big number operations. After the SSL tunnel has been established, there is no performance overhead for Nginx web server. The second application benchmark is MiniZip. MiniZip is a file compression tool that can compress the files with user specified password. In this test, we protected the password variable with SecretSafe and measured the performance overhead of the modified MiniZip by compressing a 100MB file. As we can see in Figure 5, SecretSafe protected MiniZip has only 0.2% and 1% performance overhead on x86 and x86_64 respectively. This is because SecretSafe

6. https://github.com/robertdavidgraham/heartleech

only initializes the protection code at the beginning of each execution, thus there is no performance loss during MiniZip file compression. Overall, SecretSafe enhances the application data secrecy with reasonable performance impact.

5. Related Works

In this section, we summarize the recent researches on protecting the program secret data from being leaked out.

SecretSafe protects the application secret data by fine-grained segmentation based isolation. This shares the idea of piece of application logic (PAL) isolation [10], [23], [24], [25]. Flicker [25] and TrustVisor [24] first explored how to isolate the pieces of security sensitive application logic by using a secure hypervisor. On executing the PAL code, the underlying secure hypervisor switches the execution context, making the PAL execution with memory protected. SeCage [23] also leverages the isolated VMs to protect the application secrets, and an underlying hypervisor is used to securely switch the application execution flow. A very recent work called Shreds [10] extends this concept to ARM processor with the help of ARM memory domain [41]. While those works have to involve a specialized kernel (or even hypervisor), and moreover, they require the application programmers to identify all the secret accessing code pieces. SecretSafe, on the other hand, isolates the secrets with the segmentation which has been equipped on most of the x86 processors. SecretSafe does not have to modify the OS kernel. It only requires the application programmer to identify the secrets at the variable declaration sites, and the modified compiler automatically separates the secrets and instruments the secrets accessing instructions.

Different from SecretSafe, some approaches try to store secrets inside CPU rather than on the memory chips [16], [27], [29], [30]. For example, Safekeeping [29] first breaks the private key into small pieces and disperses each pieces of the private key in memory to prevent direct information disclosure. When cryptographic computation is required, the shattered key fragments are reloaded into the x86 SSE/XMM registers to reform the private key. Thus the private key will be entirely stored in the registers only during the cryptographic computation. While for a process that does not exclusively own a CPU core, Safekeeping has to disable the interrupts and avoid entering the kernel mode when the private key is in the registers. Moreover, the CPU register based approaches could only host private keys with less than several kilobytes long [27], [30] (due to the width limitation of the CPU registers), thus they can not protect any secrets on heap as SecretSafe does. Very recently, Guan et al. proposed the Mimosa system [17] which uses the hardware transactional memory (HTM) to prevent secret keys from being illegally accessed. It leverages the strong atomicity guarantee of HTM to prevent malicious processes other than Mimosa from reading the plain text secrets. However, Mimosa has to modify the OS kernel and makes large effort in modifying the existing program code. Besides, Mimosa seems cannot be adopted by large programs, such as OpenSSL on HTM protection.

The use of segmentation has been suppressed for dozens of years, due to the widely use of memory paging. Recently, researchers have proposed several novel uses of segmentation based isolation [5], [21], [39], [40]. For example, NaCl [39] uses the segment-isolated memory region to host the critical service runtime, which implements a lightweight sandbox. Oxymoron [5] implements the page-level randomization for the shared library code by using an indirection table isolated by segment. The indirection table holds the destination of each indirect jump/call instruction, making the shared code randomization available. Code pointer integrity (CPI) [21] separates the code pointers in an isolated memory region protected by the segmentation. Therefore an attacker could not overwrite the pointer memory, hence the pointer hijacking attacks would be prevented. Although CPI is a most closely related work, CPI could not address issues such as data pointer assignment or a pointer reference to a possible secret data.

6. Summary

We have presented the design, implementation, and evaluation of SecretSafe, a lightweight and practical system to prevent the heap buffer over-read attacks. SecretSafe first employs a modified compiler to automatically separate the secrets outside of the process's logical address space. Later, a loader extension is used to load the secrets into the isolated memory, and a lightweight code monitor is set up to optimize the code execution path. With the two techniques mentioned above, SecretSafe could successfully prevent in-process secrets from being leaked out by program vulnerabilities and remote exploits. Our evaluation on real applications demonstrates that SecretSafe can prevent the buffer over-read attack with minimal costs.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work was partly supported by the National Science Foundation of China (No. 61672421, 61402358, 61602363), the Ph.D. Programs Foundation of Ministry of Education of China (No. 20120201110010) and the China Postdoctoral Science Foundation (No. 2016M590927).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Adrian Sampson. Quala: Type Qualifiers for LLVM/Clang. https://github.com/sampsyo/quala.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[5] M. Backes and S. Nürnberger. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. *Proc. 23rd Usenix Security Sym*, pages 433–447, 2014.

[6] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[7] D. P. Bovet and M. Cesati. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.

[10] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained Execution Units with Private Memory. 2016.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, , and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, August 1998.

[12] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.

[13] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):2, 2015.

[14] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[16] L. Guan, J. Lin, and B. L. Jing. Copker: Computing with Private Keys without RAM. In *Proceedings of the 21th Network and Distributed System Security Symposium*, NDSS '14, 2014.

[17] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 3–19. IEEE, 2015.

[18] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual*, Feb 2014.

[19] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, volume 47, pages 121–132. ACM, 2012.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[21] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[22] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[23] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. pages 1607–1619, 2015.

[24] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[25] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS EuroSys Conference*, April 2008.

[26] Microsoft System Center. KNOWN ISSUES FOR VIRTUAL MACHINE MANAGER IN SYSTEM CENTER 2012 R2. http://www.thomasmaurer.ch/2014/01/known-issues-for-virtual-machine-manager-in-system-center-2012-r2/.

[27] T. Müller, F. C. Freiling, and A. Dewald. TRESO: Runs Encryption Securely Outside RAM. In *Proceedings of the 20th USENIX Conference on Security*, San Francisco, CA, 2011.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[29] T. P. Parker and S. Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *International Conference on Trusted Systems*, pages 39–59. Springer, 2009.

[30] P. Simmons. Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, Orlando, Florida, 2011.

[31] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal War in Memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[32] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-edge Control-flow Integrity in Gcc & Llvm. In *USENIX Security Symposium*, 2014.

[33] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu. Risk assessment of buffer" heartbleed" over-read vulnerabilities. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 555–562. IEEE, 2015.

[34] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[35] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. SecPod: A Framework for Virtualization-based Security Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, 2015.

[36] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[37] Wikipedia. Java Performance. https://en.wikipedia.org/wiki/Java_performance.

[38] Wikipedia. Off-by-one error. https://en.wikipedia.org/wiki/Off-by-one_error.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[40] M. Zhang and R. Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 91–100. ACM, 2015.

[41] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.