Hardening the Linux Kernel Key Retention Service against Information Disclosure Vulnerabilities

Author
Elisa Guerrant

Advisor
Vasileios Kemerlis
Reader
Theophilus Benson

A thesis submitted in partial fulfillment of the requirements for earning Honors in the Department of Computer Science at Brown University

April 2019

Acknowledgements

There are many people to thank for this project. First and foremost, I would like to thank Professor Vasileios Kemerlis for his constant support in advising me, and whose generous time commitment, feedback, patience, and encouragement has been essential to this project.

I also want to thank Professor Theophilus Benson for being my official reader and, more generally, the computer science professors at Brown for being such influential role models and advisors throughout my time here.

Lastly, I would like to express appreciation for my parents John and Yolima Guerrant for motivating me and supporting my personal and academic growth.

Contents

Abstract

This thesis addresses the problem of memory safety in the Linux kernel. Despite many memory isolation, control-flow integrity, and code diversification techniques existing both in user space and kernel space, many of these techniques have been proven to be less effective in the kernel and can often be bypassed using memory disclosure vulnerabilities. This thesis describes a way in which critical data, such as keyrings that hold authentication keys, encryption keys, etc., can be moved to an isolated memory location so that, in conjunction with the kernel hardening scheme kR^X, they are protected from vulnerabilities like memory leaks. The techniques described in this paper can be generalized and used with other sensitive data in the kernel like user credentials.

1. Introduction

As more and more security measures are added to user space, many attackers are turning to the kernel as their primary target [8]. Kernel exploits are extremely powerful and efficient due to the kernel being able to run code with higher privileges than in user space and the lack of effective means of protecting data stored in kernel memory once the attacker gains access to the kernel. While some kernel hardening schemes exist, such as kernel address space layout randomization (KASLR) [4] and non-executable memory [7], this has simply shifted the types of attacks on the kernel from legacy code-injection to code reuse techniques [9]. Coupled with memory disclosure vulnerabilities, these attacks can bypass both coarse-grained and fine-grained control-flow integrity schemes and code diversification by confining hijacked control flow to valid execution paths and leveraging data leaks. The ability to disclose the memory contents of a process also leaves open the possibility of accessing sensitive, non-control data [2] such as keys, user credentials [1], and cryptographic material. In this paper, we present an approach to securing critical data structures in kernel memory utilizing the kernel hardening system kR^X [8].

2. Background

kR^X

kR^X is a kernel hardening scheme based on execute-only memory and code diversification. This scheme imposes the policy that memory can be either readable or executable, *but not both,* preventing JIT-ROP attacks [8]. While a similar scheme has been implemented in other systems using a hypervisor, memory virtualization features, or paging nuances, kR^X uses a self-protection approach that does not depend on any component more privileged than the OS kernel. It additionally utilizes a set of code diversification

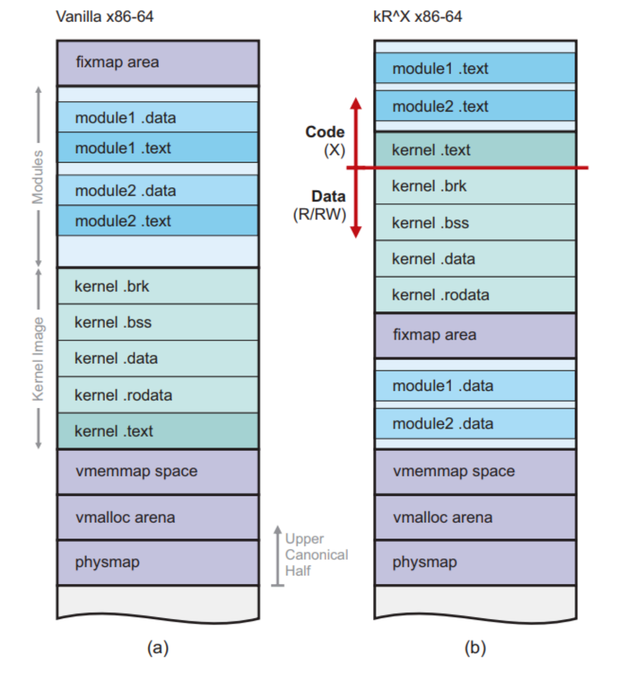

Figure 1. The Linux kernel space layout in x84-64: (a) vanilla and (b) kR^X–KAS

techniques that are written specifically for use in the kernel, dubbed fine-grained KASLR, which helps to prevent direct and indirect code-reuse attacks.

To facilitate some memory being marked as read-only or read- and write-only while code is execute-only, the kernel space layout has been changed. As Figure 1 (a) shows, the vanilla Linux kernel space layout mixes code and data regions. Enforcing a R^X policy in this configuration would be less unified and efficient. On the other hand, the kR^X kernel space layout (Figure 1 (b)) separates code and data into disjoint, contiguous regions with the code region carved from the top part of the kernel space and all other regions either left unchanged or pushed towards lower addresses. This allows R^X enforcement to be conducted much more easily and efficiently.

Kernel Key Management

One of the things we sought to protect with kR^X is the kernel key management system. Filesystems, especially remote filesystems, may require some authentication or a key to enable access; the `keyctl` API provides an interface that stores and manages this kind of information [3]. This system is divided into two parts: One part is used by the kernel to find keys for subsystems that need them. The other is accessed from and used by userspace programs to manage keys. This interface provides a system through which the kernel can access keys and userspace programs can access the add, modify, and delete operations.

Any kind of authentication or access information can be stored as a key with this interface – not just keys in the traditional cryptographic sense. The authentication or access information is stored as an opaque piece of data that is only interpreted by the kernel subsystem to which it corresponds. This design allows the interface to have far-reaching applications. Any kernel subsystem that requires authentication or access information could use it.

3. Assumptions

This project assumes that kR^X is in place as well as all of the hardening schemes used by kR^X. In particular, we assume the Linux kernel space layout defined by kR^X, which separates code and data into disjoint, contiguous regions with the code region carved from the top part of the kernel space and all other regions either left unchanged or pushed towards lower addresses. We assume an OS that implements the W^X policy in kernel space, which prevents direct (shell)code injection in kernel memory. Additionally, we presume the kernel is hardened against ret2usr attacks [6]. Any other kernel hardening feature, such as KASLR or stack-smashing protections may be supported but are not required.

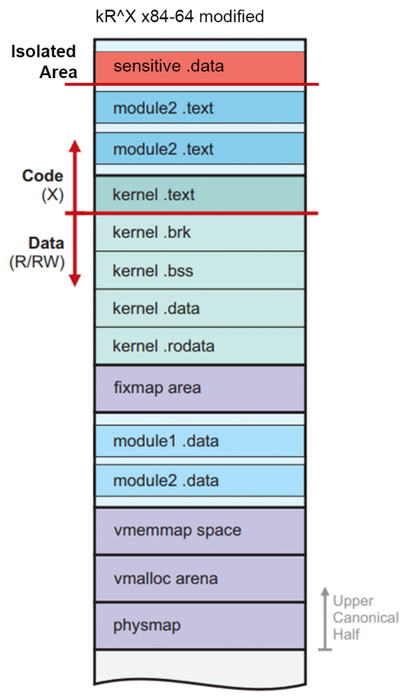

Figure 2. The Linux kernel space layout with kR^X and a new non-readable, non-writable, and non-executable region.

In addition, we assume that there has been a region carved out at the top of the kR^X kernel space layout reserved for critical data structures (shown in Figure 2) that has been marked as non-readable, non-writable, and non-executable (by arbitrary code) – i.e., it can be accessed by a few specific functions only, which are not instrumented by kR^X. This new region, like the code and regular data regions in the original version of kR^X, is both disjoint and continuous so that these permissions can be enforced in the same unified and efficient manner as in vanilla kR^X. This is the region in which memory allocated for the critical structures used by the `keyctl` interface is presumed to be located.

4. Approach

The primary factors involved in securing the `keyctl` API were identifying which data structures are allocated by the interface, which structures are critical to securing the system, and discovering how and where these data structures are allocated. To start, we both experimented with the `keyctl` interface by writing a userland program to see which pieces of information were needed to create, access, link, etc. keys. We then went through the header files defined by in the API to see which structures are defined. Section 9.1 in the Appendix shows part of the key structure, which is what is created and populated when a key is allocated. It contains several crucial pieces of information, like the description of the key, the payload, the key type, and a pointer to the structure defining the user of the key. The user structure is also a sensitive data structure since it keeps track of the identities of each user and information about which keys they own.

We then went through the code in the API to determine where pieces of data are allocated and freed. This allowed us to gain more information about which data structures are allocated while completing key operations, in particular structures that are not specific to the `keyctl` interface which nonetheless contain sensitive information, and the method through which the memory for these data structures is allocated. Besides the key and user structures, in several instances the `keyctl` interface makes use of buffers to pull pieces of information from user space into kernel space. Pointers to these buffers are saved in key structures so it is imperative that these are included in the list of critical data structures that need to be moved to a non-readable location. On the other hand, buffers populated with information that is to be returned to the user like in the describe operation can be left alone.

Besides deciding which data structure are crucial, reading and analyzing the code allowed us to see how memory was allocated throughout the `keyctl` interface. The interface primarily makes use of a combination of `kmalloc`, lookaside caches (which are implemented using `kmalloc`), and string routines (which also use `kmalloc`). This indicates that the data structures used by the `keyctl` interface use memory from the `physmap` region. In the kR^X kernel space layout, this region is readable so the structures must be allocated in another way.

5. Implementation

The project was coded on the Debian 3.19.0 operating system configured with SysV init [5] and to be compatible with kR^X.

In implementing the necessary code changes described above, we began with reimplementing the functions `strndup_user`() and `memdup_user`(), since all other code changes would be directly in the `keyctl` interface files. Both were used throughout the `keyctl` API and used `kmalloc` to allocate memory. Section 9.2 of the Appendix shows these functions. These rewritten functions replaced all instances of the originals in the `keyctl` interface since they were used to pull user-supplied pieces of data into kernel space. After this change (and all other changes to how memory was allocated within the API), we went through and found all locations in which the memory was freed and changed the function used to free the memory from `kfree`() to `vfree`()where appropriate.

The next step was to find all instances in which user and key structs and the different pieces of data they contained, like payload and description, were allocated and freed then replacing those function calls with `vmalloc`()/`vfree`(). The majority of the code changed for this part was in the files `keyctl.c` and `key.c` but there were also a few scattered instances throughout the rest of the API. This process of identifying memory allocations and frees then replacing them if they involved one of the critical data structures identified earlier continued for all of the files in the `keyctl` API.

After making these changes, the cache titled `key_jar` which had been used by the `keyctl` interface to manage key memory allocations was no longer necessary so we removed where it was declared and instantiated.

Limitations

The choice to use `vmalloc` directly throughout the `keyctl` API has some shortcomings. For one, `vmalloc` is slightly less efficient than `kmalloc` and some other means of memory allocation. On a system in which this API is used frequently, and particularly a system in which keys are repeatedly created and destroyed, this difference in speed could significantly impact the efficiency of the programs utilizing the key management system. Additionally, `vmalloc` can only allocate memory in full pages. In many instances, the amount of memory needed to allocate, say, a key structure or user structure will not be close to filling a whole page, leading to a large waste of memory. On a smaller scale, this does not have a significant impact on the functionality of the API. Having said that, if a large number of keys needed to be allocated at once, the program could easily run out of memory.

Since several parts of the original `keyctl` interface used lookaside caches and a slab allocator (which allocates memory using `kmalloc`), one way forward would be to implement a version of a slab allocator that uses `vmalloc` instead of `kmalloc`. This would significantly improve the memory waste problem that using `vmalloc` directly presents.

6. Evaluation

The most important aspect of the project was ensuring that it was possible to move the critical data structures used by the `keyctl` interface while preserving the integrity and functionality of the interface. To do this, each key operation needed to be tested and stressed. This included testing instantiating, updating, reading, revoking, destroying, and requesting keys, among other operations. We also sought to ensure that the keys that were allocated contained the correct information (e.g. the keys allocated had the correct uid, gid, description, etc.). Our primary focus within this was to assure creating and destroying both keys and users had not been affected since these functions had the most code changed and are where the majority of the `keyctl` data is created and destroyed.

To do this, we wrote a user space program which utilized the `keyctl` API. Since the interface pushes most of the policy and management pieces to userspace, we was able to write a program which tested all of the different key operations. To ensure that allocating and destroying keys worked correctly, we wrote extra cases in which we allocated large numbers of keys, scheduled keys for destruction by both unlinking them and revoking them, attempted to use keys that had already been revoked, created keys of a user-defined type, etc. in order to test all parts of the key allocation and destruction worked as expected. If too many keys were allocated at once, the program did run out of memory but this is, of course, due to the fact that `vmalloc` allocates full pages rather than smaller pieces of memory.

7. Related Work

This thesis works off of the ongoing kernel hardening scheme project kR^X by Marios Pomonis et al. described in Section 2. PrivWatcher [1] is another system which protects non-control data in the kernel. This system, however, does not utilize a self-protection approach like kR^X and instead requires an isolated execution domain in which PrivWatcher mediates changes to non-control data by the kernel.

8. Conclusion

While there are some short comings with the approach illustrated in this paper, chiefly the fact that using `vmalloc` directly within the `keyctl` interface wastes a lot of memory, this project serves as a proof-of-concept, demonstrating the feasibility of moving critical data structures out of the `physmap` region of kernel memory and into an isolated memory location and serves as a guide for how other interfaces can be altered to use virtual memory allocation rather than `kmalloc`. This project in conjunction with the altered kR^X makes it so that an attacker cannot read sensitive data allocated by the `keyctl` interface from an attack utilizing code reuse techniques.

9. Appendix

9.1 Key Struct

```
struct key {
      atomic_t          usage;           /* number of references */
      key_serial_t            serial;          /* key serial number */
      union {
            struct list_head graveyard_link;
            struct rb_node    serial_node;
      };
      struct rw_semaphore     sem;
      struct key_user         *user;           /* owner of this key */
      void              *security;  /* security data for this key */
      union {
            time_t            expiry;          /* time at which key
expires (or 0) */
            time_t            revoked_at; /* time at which key was revoked
*/
      };
      time_t                  last_used_at;
      kuid_t                  uid;
      kgid_t                  gid;
      key_perm_t        perm;       /* access permissions */
      unsigned short          quotalen;   /* length added to quota */
      unsigned short          datalen;

...

/* the key type and key description string */
      union {
            struct keyring_index_key index_key;
            struct {
                  struct key_type   *type;           /* type of key */
                  char        *description;
            };
      };

  /* key data
       * - this is used to hold the data actually used in cryptography,
etc.
       */
      union {
            union key_payload payload;
            struct {
                  struct list_head name_link;
                  struct assoc_array keys;
            };
            int reject_error;
      };

...

};
```

9.2 Rewritten `util.c` Functions

```
void *memdup_vmalloc(const void *src, size_t len)
{
        void *p;

        p = vmalloc(len);
        if (!p)
            return ERR_PTR(-ENOMEM);

        memcpy(p, src, len);

        return p;
}

char *strndup_vmalloc(const char *s, long n)
{
        char *p;
        long length;

        length = strnlen_user(s, n);

        if (!length)
            return ERR_PTR(-EFAULT);

        if (length > n)
            return ERR_PTR(-EINVAL);

        p = memdup_vmalloc(s, length);

        if (IS_ERR(p))
            return p;

        p[length - 1] = '\0';

        return p;
}
```

10. References

[1] Azab, A.M., Chen, Q., Ganesh, G., and Ning, P., 2017. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. ACM.

[2] Chen, S., n.d. Non-Control-Data Attacks Are Realistic Threats. Purdue University.

[3] Edge, J., 2006, November 21. Kernel Key Management. Accessed April 05, 2019. https://lwn.net/Articles/210502/

[4] Edge, J., 2013, October. Kernel address space layout randomization. Accessed April 10, 2019. https://lwn.net/Articles/569635/

[5] init: SysV-style. https://en.wikipedia.org/wiki/Init#SysV-style

[6] Kemerlis, V.P., G. Portokalidis, and A. D. Keromytis, 2012.. kGuard: Lightweight Kernel Protection against Return-touser Attacks. Proc. of USENIX Sec, pages 459–474.

[7] Larkin, M., 2015. Kernel W^X Improvements In OpenBSD. Hackfest.

[8] Pomonis, M., et al., 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. ACM.

[9] Shacham, H., 2007. The Geometry of Innocent Flesh on Bone: Return-into-libc without Function Calls (on the x86). ACM.