Reviewing “From Collision To Exploitation”

Julian Ganz
Department of Computer Science
Furtwangen University
Furtwangen, Germany
Julian.Ganz@hs-furtwangen.de

Matthias Beyer
Department of Computer Science
Furtwangen University
Furtwangen, Germany
Matthias.Beyer@hs-furtwangen.de

***Abstract*—In their paper "From Collision to Exploitation", Xu et al present a strategy for exploitation of use-after-free vulnerabilities in the Linux kernel. In this paper, we review these exploitation attempts and evaluate how applicable they are. We try to reproduce a use-after-free attack after introducing an artificial echo kernel module. Also, we measure the success rates of our exploit and describe some mitigation techniques.**

***Index Terms*—Memory Collision; Use-after-free Vulnerability; Linux Kernel Exploit; Review**

I. Introduction

In their paper “From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel”, Xu et al. present a strategy for exploitation of use-after-free vulnerabilities in the Linux kernel [1]. The strategy was illustrated using two distinct attacks on a vulnerability introduced via a kernel module specifically for demonstration purposes. Additionally, both an evaluation of their attack and mitigation strategies were provided.

In their paper, they present high success rates for both of their attacks. They also refer to contemporary versions of the Android operating system, claiming their attacks can be leveraged for rooting smart phones running these versions.

In this paper, we shortly recap the attack strategies presented in the original paper. Like Xu et al., we introduce a character echo device featuring an artificial use-after-free vulnerability. Furthermore, we describe our attempts to reproduce the object based attacks in a low-noise environment while targeting the echo device. We present our findings regarding the reproducibility of the aforementioned attacks as part of an independent evaluation.

We conclude with an assessment of the mitigation techniques proposed by Xu et al. and proposals for future work.

II. Vulnerability

The attacks presented operate on use-after-free vulnerabilities. A use-after-free is a programming error often encountered in C and old C++ codebases. In those languages, developers use an allocator, sometimes directly, to acquire memory to store data. If the memory is no longer needed, the developer frees the allocated memory, making it available for future use, e.g. by other modules of the same program. The allocator may now return that memory range to an allocation request. The term use-after-free denotes the process of using objects or data after the underlying memory was freed. [2] Since the allocated memory may be read or controlled by an user or other party, a use-after-free is a potential vulnerability.

A. Kernel vulnerabilities

The attacks presented are targeted at the Linux kernel. As an operating system kernel, it is responsible for process scheduling and basic access control as well as management of hardware and other resources [3, sec. 1.3]. Kernel exploits generally have a high criticallity, since the kernel controls the hardware and all processes running on a computer. Hence, if an attacker can control the kernel, he or she can also control all software running on the kernel. Even limited control can be used for manipulation of processes or privilege escalation by an attacker.

The kernel is, however, a heavily multi-threaded, deterministic program, driven by numerous sources

of input [3, sec. 2.4]. For example, each process running on the system interacts with the kernel by simply running[1]. Unlike in a single-threaded program, exploiting a use-after-free is not trivial. Reuse of memory previously occupied by a now freed object can not only be triggered by the attacker but also by other processes or the kernel itself.

The behavior of the allocator used in the vulnerable code must also be considered when trying to exploit an use-after-free. In the kernel, the slab allocator is often used for allocating objects and buffers. The slab allocator maintains a "cache" for each type of object to be allocated as well as generic caches, e.g. for user buffers. Each cache manages memory areas of a predefined size, which are grouped in "slabs". [3, sec. 12.6]

Unsurprisingly, Xu et al. also discussed the slab allocator in their paper [1].

B. Recent Kernel Vulnerabilities

In 2016, several use-after-free vulnerabilities were found in the linux kernel, which could be used by an attacker to exploit the linux kernel for example to gain privileges.

- CVE 10088 was found to be a use-after-free vulnerability "sg" implementation in linux kernel 4.9. It allowed users to read arbitrary kernel memory or cause a denial of service [4].
- CVE 8655, a race condition which lead to a use-after-free bug in the network stack allowed an attacker to cause a denial of service [5].
- CVE 7117, also a use-after-free bug in the network stack affected all kernel versions prior to 4.5.2 and gave an attacker the possibility to execute arbitrary code in the kernel context [5].

C. Kernel module

For demonstration purposes, Xu et al. developed a kernel module, which introduced a vulnerable system call [1]. It would take as argument two integer, "opt" and "index" and carry out one of three actions, which could be selected by the former argument:

- allocate memory for code to execute. A pointer pointing to that memory would be stored in an array.
- free the memory pointed to by the pointer stored in the array at the given "index".
- execute the buffer pointed to by the array element at "index" as if it were a function.

The code presented in their paper was not reused for reproduction for a number of reasons. For example, the code appeared to be incomplete. A few of the symbols used could not be found in the Linux kernel sources. Also, adding a system call the way presented was deemed "hacky", since system calls are usually not added by kernel modules at runtime but added to a file in the kernel sources, from which a static table is generated at build time.

For drivers, on the other hand, facilities exist for loading and set-up during run-time. Also, drivers of various type are far more often added to the kernel than system calls, through which programs interact with those drivers and may thus contain and expose far more vulnerabilities. Hence, we developed a vulnerable character device for reproducing the attacks proposed by Xu et al.

III. Object-based attack

The first attack described by Xu et al. exploits the characteristic of the slab allocator to return memory areas from partially filled slabs first. The goal is to push an object with controlled contents to areas of freed objects which are still referenced and then trigger the reuse of the object, carrying out an action which is at least partially controlled by the attacker. Classically, this is achieved by:

- causing the kernel to allocate an object to be targeted
- causing the kernel to free the object, leaving it vulnerable
- pushing an object with controlled contents to the position of old object
- triggering the use-after-free

Note that such an attack heavily relies on a collision in memory between the old, vulnerable object and the new contents, which is pushed by the attacker. For a collision, the freed area has to be the next in turn to be returned upon an allocation request. This is not likely, since any other process or kernel thread might be scheduled during the attack and cause other slots to become free or the targeted slot to be occupied again upon another request.

[1] since it has to be scheduled

For this reason, Xu et al. allocate not one but a large number of objects, which are then freed. Also, many objects are pushed by the attacker before triggering the use-after-free for all the targeted objects. By raising the number of both targeted and pushed objects, the probability of achieving a collision naturally rises, too.

Still, the attack is limited regarding the objects which can be targeted, since a collision can only be achieved with objects in the same cache. Also, the control over these in-kernel objects is often limited. Buffers filled with contents from user-space often reside in general purpose caches for different sizes. However, these are rather rarely used for anything but buffers with varying size. Hence, we expect them to be of limited use when attacking kernel code.

As Xu et al. explain, however, an attacker can overcome these limits when he or she can arrange for an entire slab to be emptied, e.g. by causing large numbers of targeted objects to be freed. In this case, the slab itself may be de-registered, freeing the page(s) it previously occupied in memory. The now free page(s) can be reused for other purposes, including, of course, another slab. As this slab may be part of another cache, an attacker can overcome the aforementioned restrictions. When crafting an attack, the differences in memory layout has to be respected. On the other hand, a payload may overwrite many targeted objects at the same time.

IV. Physmap-based attack

When looking at the x86-64 architecture, we can see that 48-bit virtual adresses that are sign-extended to 64-bit addresses. This splits the virtual address space into two 128 TB chunks, where the kernel-space resides in the upper half and the lower half is reserved for user-space pointers.

The kernel address space is divided into six regions

- The *fixmap* area
- modules
- Kernel image
- *vmemmap* space
- *vmalloc* arena
- *physmap*

[6]

The *physmap* area of the kernel maps dynamic kernel memory allocation. In the kernel there are two ways to allocate memory: *vmalloc* and *kmalloc*. Both of these variants of kernel memory allocation must hold a number of properties, such as continuity of virtual or physical memory, timing constraints and should guarantee to succeed. Given these constraints, the *physmap* is a main factilitator of performance [6].

The *physmap* is architecture-indenpendent. It is always located at a known adress, even if Kernel Adress Space Layout Randomization (KASLR) is enabled. Although, its exact location can differ between architectures [6].

Xu et al. show an Physmap-based attack where they copy a shellcode to the kernel and trigger a use-after-free in their kernel module to execute this shellcode. They do so in seven steps:

1) Allocating memory for padding
2) Allocating vulnerable memory
3) Freeing the padding memory
4) Freeing the vulnerable memory
5) Writing their shellcode to a “mmap” buffer
6) (Locking the buffer)
7) Triggering the use-after-free in their kernel module which then executes the buffer allocated beforehand

V. Echo character device

For our attack, we implemented a character device which represents a echo-device working with a ring-buffer internally. For simplicity, the ring-buffer is implemented as array of message buffers (1).

Source Code 1: Message Buffer Structure

```
struct msgbuf {
    short size_of_msg;
    short copied_to_user;
    char  msg[1];
};
```

An user can write to the ring-buffer in the kernel module by writing to the character device mounted at “/dev/vulnchrdev”, which is automatically registered by the kernel module upon loading.

The device implements a “dev_write()” function which gets called by the kernel whenever a program writes to the device, for example via the “echo” command. The “dev_write()” function allo-

cates message buffers in the ring-buffer to hold the user-supplied content using “kmalloc()” (2).

Source Code 2: “dev_write()”

```
// ...
struct msgbuf* msg;
size_t l = sizeof(msg) + len;
msg = kmalloc(l, GFP_KERNEL);
snprintf((char*) &msg->msg,
    len, text);
/* more message setup */
*w_msg = msg;
w_msg++;

// wrap write-ptr
if (w_msg > msgs + NBUF)) {
    w_msg = msgs;
}
// ...
```

If an user reads a message from the ring-buffer, the “dev_read()” function gets called in the kernel module. This function calculates the number of bytes which can be copied to the user. This number is either limited by the size of the buffer supplied by the user, the number of bytes in the message or the number of bytes which are not yet copied to the user (3.

Source Code 3: Copying message to user

```
// ...
ncpy = (*r_msg)->size_of_msg
     - (*r_msg)->copied_to_user;
ncpy = nbuf < ncpy ? nbuf : ncpy;

nerr = copy_to_user(bffr,
    (&(*r_msg)->msg) +
    (*r_msg)->copied_to_user,
    ncpy);

(*r_msg)->copied_to_user += ncpy;
// ...
```

The message gets freed as soon as it is copied to the user entirely. If the user does not supply a large enough buffer for copying the message to her, the message is marked as partially copied. Later, the remaining part of the message can be read by the user.

The artificial bug in our kernel module is a non-existent check whether the reader for the ring-buffer overtakes the writer, thus leading to reading non-allocated or already-free’d memory. With this bug in place, one can read arbitrary kernel memory into userspace, thus fetching sensitive data like passwords or cryptographic keys.

VI. Reproduction

For reproduction, we built a minimal custom kernel 4.9 with very few drivers as well as a minimal root image. We used `dietlibc-0.33` [7] as the libc for an ext4 root image featuring:

- the `embutils-0.19` core utils [8],
- the `sash-3.8` stand alone shell[2] [9],
- the `kmod-23` module management [10],
- a minimal `init` program,
- a minimal `halt` program,
- the kernel module featuring the echo character device described in section V,
- exploit implementations and
- a minimal test script, which loads the module, runs an export and halts the virtual machine afterwards.

The kernel was run using `qemu-2.8` [11], supplying the `-kernel` and `-append` options. The root image was bound as a “virtio” virtual block device. For this purpose, the corresponding driver was built into the aforementioned minimal kernel.

We choose to use a minimal setup for testing the exploits in order to reduce noise which could potentially interfere with our exploit. Specifically, the Virtual Machine (VM) was not connected to any network. Only a virtual serial console was used to “talk” to the machine.

A. Payload

As payload, we use a message of the exact same size as the messages sent over the echo device, with an appropriate header. Hence, the size of the payload is equal to the size of the buffers used internally in the echo device.

Xu et al. described pushing their payload into kernel-space using the `sendmmsg()` system call [1]. The system call accepts message headers containing a destination, message contents and control data. [12], [13] The latter was used by Xu et al. for transporting the payload since the control data is copied to a `kmalloc()`’d buffer as apparent from Source Code 4.

[2]patches were required for building sash with `dietlibc`

We use the `sendmsg()` system call instead, which does only send a single message. However, aside from the number of messages sent, the system calls are similar concerning the handling of control data. E.g. both use the function illustrated in Source Code 4.

Source Code 4: Excerpt from `net/socket.c`

```
static int ___sys_sendmsg(...) {
  // ...
  if ((...) && ctl_len) {
    // ...
  } else if (ctl_len) {
    if (ctl_len > sizeof(ctl)) {
      // sock_kmalloc wraps kmalloc()
      ctl_buf = sock_kmalloc(sock->sk,
        ctl_len, GFP_KERNEL);
      if (ctl_buf == NULL)
        goto out_freeiov;
    }
    // ...
  }
  // ...
}
```

Xu et al. didn't mention in their paper the requirements on the payload size imposed by the aforementioned listing. A buffer is not allocated unless the control data exceeds a certain limit. Otherwise, a statically allocated buffer is used. While this detail is irrelevant for illustrating the exploitation strategies described in their paper, it has to be considered when designing an actual attack.

It is also important to note that not any arbitrarily large size can be chosen for the control data, since it is limited per socket. Although this limit is configurable, attackers usually lack the privileges necessary.

Another aspect which did receive little coverage in their paper was the restrains on the control data itself. For an `AF_UNIX` socket, for example, the control data is interpreted by the kernel and must therefore satisfy a specific format: the control data is expected to consist of a series of control messages, each with a header. [14]

Both our vulnerable echo device and the control message header start with a field containing the message size, as a **`size_t`**. However, while the control message header's size-filed denotes the size of the entire message [15], the echo device's size-field denotes the size of only the message received from the user. To avoid `sendmsg()` from failing, we set the size correctly for the control message header, potentially causing `read()`s to return a message a few bytes longer than in the original payload.

The control message header has additional fields by which the scope of a message is identified. Taking no measures regarding the additional control header fields did not cause `sendmsg()` to fail, since the additional fields happen to be zeroed by our payload. Interestingly, control buffers are still copied to kernel memory even if the control headers are invalid in length, causing the system call to return an error. However, the control data in the kernel is deallocated after the processing of each message, regardless of whether it is valid or not[3].

Hence, sequential pushes of a buffer via `sendmsg()` or `sendmmsg()` will use the same SLAB slot with high probability, if the SLAB is not used otherwise. This obviously impedes pushes of payloads to as many SLAB slots as possible in order to increase the collision probability as described in section III.

Is is important to note that, since the SLAB slot is released before the system call returns, the payload may not only be overwritten by new payloads but also by other contents. Hence, the situation is rather unstable. More persistent kernel buffers would be far more advantegous. However, we felt that searching for a possibility to push such a buffer would be outside the scope of this paper.

For example, parts of control data *might* be copied to other kernel buffers for some socket types during a `sendmsg()` call. Other system calls, interfaces or character devices might be leveraged in order to push a payload into a controlled, persistent buffer.

B. Object based attack

For reproduction of the object based attack, we developed an exploit operating on the vulnerable echo device. It uses pairs of anonymous connected sockets for pushing the payload. By never reading those messages, we ensure that the control data is kept in memory. Originally, our exploit imitated the behaviour described by Xu et al., it

1) opened the echo device,

[3]at least in kernel 4.9

2) produced objects by writing messages,
3) read those messages back, freeing the kernel objects,
4) pushed payloads via `sendmsg()` and
5) triggered use-after-free by reading the messages a second time from the echo device.

As described in subsection VI-A, memory is allocated from the same SLAB slot repeatedly in such a situation. We suspect that the code presented by Xu et al. is only an abstraction of the original exploit. We considered using parallel calls to `sendmsg()`, but decided to simply reorder the actions taken by the exploit instead: right after an object is released in the third step, we push several payloads via `sendmsg()`.

In low noise environments, a payload will still most likely be overwritten with a new one. However, since at least the message we read from the echo device is released, each new batch of payloads encounters a new situation regarding the SLAB usage. Therefore, the payloads are spread across multiple slots with a higher probability.

We can switch between the two object based attack strategies described by Xu et al. [1] via a command line parameter. That command line parameters controls the number of slots in a SLABs targeted. If "1" is passed, one payload is sent as control data and we are performing the first of the object based attacks described in "From Collision To Exploitation". It will potentially collide with a vulnerable object if the buffer is allocated in the same SLAB cache, e.g. if the size is identical to the size of the targeted buffer.

If a number greater than "1" is passed, multiple payloads are sent as control data. The payloads are aligned to sizes of slots in a targeted SLAB. If an entire SLAB is freed as described by Xu et al., one single of those buffers may collide with the entire SLAB targeted, given that a sufficiently high number was passed via the command line. This way, we can achieve multiple collisions with a single buffer pushed.

C. Physmap based attack

The physmap based attack was not reproduced because of limited time. The mechanism used is not impeded by the need of abusing a system call for pushing a payload to a buffer in kernel space. Instead, regular `mmap()` calls are used for overwriting the vulnerable objects. As `mlock()` calls are used to keep the payload in RAM, the approach also doesn't suffer the drawbacks illustrated in subsection VI-B. Hence, we deem the physmap based attack superior to the object based attack, given that a collision can in fact be achieved.

D. Success rates

We used the aforementioned minimal setup for automated testing of the exploit described in subsection VI-B. We chose a small set of values for some parameters:

- "Writes per freed object" denotes the number of times a buffer containing payload is pushed for every vulnerable object.
- "Number of payloads per buffer" denotes the number of payloads pushed in one `sendmsg()`.
- "memory" denotes the memory given to the virtual machine.

For each of the configurations possible given the values chosen, 100 runs were performed and the results logged. The percentage of kernel buffers affected was extracted from each logs. Tables were generated which map a configuration to the number of runs in which exactly a specific percentage of buffers was successfully overwritten by an attack.

Table I illustrates the success rates for object based attacks targeting single slots, e.g. "Number of payloads per buffer" is 1. In our low noise environment, 27% or 28% of all the vulnerable buffers were overwritten by the exploit. Hence, all the runs successfully pushed payloads to the kernel buffers targeted, as is to be expected in a low noise environment.

For object based attacks targeting entire SLABs, however, the situation is different. For configurations pushing more than one payload per buffer, no single attack was successful. We can assume that an entire SLAB was freed because we create 100 vulnerable kernel objects and our targeted message requires slots of 128 Bytes in size. From `/proc/slabinfo` we know that SLABs in the relevant cache contain 32 objects. Hence, three SLABs have to be stripped bare of objects in a run, assuming that no other actor allocated buffers in the meantime. During the runs performed, buffers

containing 32, 64 and 128 objects were pushed, filling out entire pages.

However, as explained above, buffers pushed by our exploit have a high potential to re-use the exact same slot repeatedly. Hence, instead of allocating a new page and achieving a collision, an existing SLAB is (re)used. Therefore, this attack is unlikely to work without persistent kernel buffers.

VII. Mitigation

To mitigate the impact of the attack described above, Xu et al. propose several mechanisms.

First, Xu et al. propose to limit the percentage of total memory one user can allocate on the system. If that percentage is hit, the kernel refuses to allocate more memory for the users processes. They argue that limiting the amount of allocateable memory for each *process* instead of each user is not feasible, as all active processes can influence the kernel memory, thus, an attacker can create several coordinated attack processes to allocate kernel memory. Such a functionality could be implemented using the "cgroups" feature of modern linux operating systems [16]. "cgroups" can be used to limit resources per-process, thus restricting an user to a certain number of processes and limiting each process to (for example) 100 MiB maximum RAM usage.

Table I: Objects affected by object based attack 1

writes	mem	$< 27\%$	27%	28%	$> 28\%$
1	64M	0	69	31	0
2	64M	0	63	37	0
3	64M	0	54	46	0
5	64M	0	61	39	0
10	64M	0	65	35	0
1	128M	0	89	11	0
2	128M	0	93	7	0
3	128M	0	92	8	0
5	128M	0	95	5	0
10	128M	0	92	8	0
1	256M	0	94	6	0
2	256M	0	86	14	0
3	256M	0	91	9	0
5	256M	0	93	7	0
10	256M	0	90	10	0
1	512M	0	94	6	0
2	512M	0	90	10	0
3	512M	0	90	10	0
5	512M	0	87	13	0
10	512M	0	90	10	0

This table shows the number of runs in which exactly the given percentage of kernel buffers were overwritten by an attack, for each configuration.

Their second idea proposes to seperate the kernel memory SLAB caches and the physmap entirely, thus only preventing the physmap-based attack. They rate the effectiveness of this defending mechanism as high, as the attack heavily relies on the present memory layout. However, they argue that implementing such a separation is hard, as on 32-bit kernels the memory address space is too small.

Mechanisms like KASLR could possibly complicate further exploitation of the attack presented in this paper.

VIII. Future work

In this paper, we described our reproduction of the object based attacks proposed by Xu et al., in a minimal setup with minimal noise. Because of limited time we did not reproduce the physmap based attack. Reproducing it should be in the focus of future work.

As laid out in section VI, using the control data of a message for pushing the payload via `sendmsg()` appeared to be not ideal. In the future, a better solution should be investigated for carrying out object based attacks.

Once both the object based and the physmap based attacks are reproduced, their performance should be measured in a more noisy environment. For example, a controlled noise agent could be introduced in the minimal setup. It would cause allocation and deallocation of kernel objects repeatedly, simulating network traffic or other noise. The measurements performed for this paper could then be repeated or different noise levels.

A. An exploit strategy library

A working reproduction which is still functional under reasonable noise levels could be beneficial to practical security research. The only component which has to be replaced in an exploit for adaption to another use-after-free vulnerability is the payload or a generator for the payload. Hence, the strategy could be abstracted by a library providing a single function. That function would take as an argument the payload or a generator and carry out the attack. It would abstract architecture specific information

and lay out the buffer based on slab sizes known or provided by the user.

B. Mitigation using cgroups

In VII, we already hinted at cgroups as a potential possibility for mitigation of the attacks described. Researching their actual potential for mitigation also remains future work.

IX. Conclusion

In this paper we described reproduction attempts of some of the attacks described and assessed the mitigation techniques proposed by Xu et al. As target for our attack, we used an artificial vulnerability within a character echo device which was developed for that purpose.

Some flaws with the approach chosen for pushing payloads into kernel space in the context of the object based attack were identified and described. While some details not mentioned by Xu et al. were be ascertained, the behavior of the exploit could not be reproduced exactly as advertised. Hence, we have to assume that some details important for reproduction are still unknown to us. Nonetheless, we were able to reenact the attacks described by Xu et al. and achieved high success rates for some modes of operation using our version of the exploit in a low noise environment.

The physmap based attack was not reproduced. However, we assume that the attack's performance is similar to that of the object based attack, as shown by Xu et al.

The exploits described in subsection VI-B were not tested in a noisy environment. However, an experimental setup for measuring the performance in a noisy environment was proposed in this paper.

We assessed the mitigation techniques described by Xu et al. and agreed with their view. Furthermore, we proposed using cgroups for implementing the proposed measures.

Acknowledgment

We want to generally thank the authors of man pages. We find writing software much easier if you don't have to rely on web search results for high quality documentation. We especially want to thank the authors of the man pages cited in this paper.

References

[1] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 414–425. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813637

[2] "Cwe-416: Use after free." [Online]. Available: https://cwe.mitre.org/data/definitions/416.html

[3] R. Love, *Linux kernel development*, 3rd ed. Boston, Mass.: Addison Wesley, 2010. [Online]. Available: http://proquest.tech.safaribooksonline.de/9780768696974

[4] "Cve-2016-10088." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10088

[5] "Cve-2016-8655." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655

[6] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 957–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kemerlis

[7] F. von Leitner, "diet libc – a libc optimized for small size." [Online]. Available: http://www.fefe.de/dietlibc/

[8] ——, "embutils – small system utilities for embedded systems." [Online]. Available: http://www.fefe.de/embutils/

[9] D. I. Bell, "sash – a stand-alone shell with many built-in commands." [Online]. Available: http://members.canb.auug.org.au/~dbell/

[10] "kmod – module management." [Online]. Available: https://git.kernel.org/cgit/utils/kernel/kmod/kmod.git

[11] "Qemu open source processor emulator." [Online]. Available: http://qemu.org/

[12] *sendmsg(2) Linux Programmer's Manual*, March 2016.

[13] *sendmmsg(2) Linux Programmer's Manual*, March 2016.

[14] *unix(7) Linux Programmer's Manual*, July 2016.

[15] *cmsg(1) Linux Programmer's Manual*, March 2016.

[16] *cgroups(7) Linux Programmer's Manual*, March 2016.