A Survey of General Purpose OS Security Vulnerabilities

COMP5900T OS Security Assignment

by

February 20, 2020

Abstract. In this paper, we will examine four vulnerabilities in four distinct general purpose[1] operating systems – three open source and one closed source. We will discuss the severity and details of each vulnerability, and conclude with a broader conjecture about the differences between open and closed source operating systems with respect to security assurances. Ultimately, we find that although general purpose OS vulnerabilities are often quite severe, they are in many cases inevitable; but the open source model is far more conducive to open and constructive discussion for fixing and preventing these vulnerabilities in the future.

1 Introduction

General purpose operating systems are complex; while this complexity is largely necessary to support a wide variety of software and hardware, it consequently hinders efforts to provide security assurances to these systems [13]. This inverse relationship between complexity and security assurances is compounded by the nature of security as a negative goal [22]; that is, it is much easier to show that a system is *unsecure* than to show that it is *secure*. This notion of complexity and limited verifiability generally results in the unintentional introduction of vulnerabilities into general purpose operating systems.

In addition to the aforementioned complications, further distinctions arise when considering the nature of open source and closed source systems with respect to their security. Through

[1]Here we define a general-purpose operating system as one designed to run a variety of end-user applications on generic hardware; i.e. the opposite of a task-specific operating system.

the analysis of security vulnerabilities in four distinct operating systems, we will discuss the relationship between a system's complexity and the verifiability of its reference monitor. Finally, we will argue that closed source operating systems are not necessarily more secure than open source operating systems, despite common misconceptions therein.

2 Linux: Realtek Wifi Driver Remote Code Execution

In October of 2019, a vulnerability in the Realtek wifi driver for the Linux kernel was discovered by Nico Waisman of the GitHub security lab [1, 3]. This vulnerability was quickly reported by all major Linux distribution vendors[2] including Arch [2], Red Hat [27], Debian [9], Canonical (Ubuntu and variants) [3], and SUSE [29]. This vulnerability affected all kernel versions prior to 4.19.82 LTS or 5.3.6 [2, 20].

The exploitable code resided in the driver code for Realtek WiFi devices. In particular, two functions failed to check the upper bound of a variable, `noa_num`, which was responsible for accessing an array within the `p2pinfo` data structure [1]; as a result, specially crafted input would allow an attacker attempting to connect to a device via peer-to-peer to cause a buffer overflow in kernelspace.

2.1 Buffer Overflow Attacks

Buffer overflows are a class of attack wherein an attacker (usually with the help of specially crafted input) is able to escape the range of memory assigned to a buffer and thus overwrite unintended locations within an address space [4, 22]. While we may generally be tempted to think of buffer overflow attacks in the context of userspace, these attacks are also present in kernelspace, and invariably comport with a much higher risk to the system. For example, if the attacker is able to use the buffer overflow to achieve arbitrary code execution, they are able to operate freely in ring 0, with full access to the entire system. Additionally, buffer overflows present a perfect vector for a myriad of other attacks, including escalation of privilege, tampering with system functionality (including that of the TCB[3]), and denial of service [13, 22].

[2]This list is only partially complete.
[3]Trusted computing base.

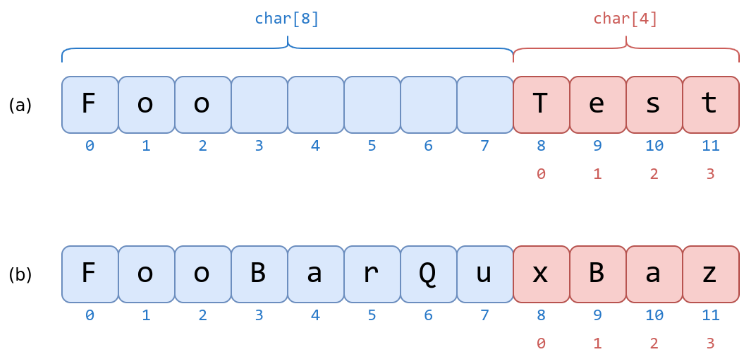

Figure 2.1: A depiction of a simple buffer overflow[4]. Consider two buffers[5], one of size 8, and one of size 4, located next to each other in memory. (a) depicts the result of writing `Foo` to the first buffer, and `Test` to the second. (b) depicts the result of appending `BarQuxBaz` to the first buffer, such that it overflows the allocated memory for the first buffer and overwrites the second.

Figure 2.1 depicts a simplified example of buffer overflows. Although we consider two adjacent character buffers in the example, in practice we are often interested in overwriting critical locations in memory, such as return addresses [28]. Additionally, protection mechanisms such as address space layout randomization (ASLR) and stack canaries [22] complicate the exploit process significantly compared to what is shown. Although memory protections do exist in kernelspace [15], such invalid access attempts generally cause a kernel panic, which requires that the system be rebooted in order to return to a sane state – the primary consequences here are denial of service to the user(s) of the system. In cases where the invalid access attempt is *not* caught, the consequences can be much worse. We discuss the implications of this in further detail in Subsection 2.2.

2.2 The Vulnerability in Detail

In order to exploit the vulnerability in the Realtek wifi driver code, an attacker takes advantage of a power-saving feature [2] called Notice of Absence. In particular, the attacker can send a Notice of Absence number (`noa_num`) of arbitrary length, due to insufficient checks on its upper bound in two instances. Since this number is used to index into several buffers contained within a kernel data structure, this leads to a buffer overflow as described in the previous subsection.

The vulnerable conditional, shown in Listing 2.1, checks to ensure that `noa_len - 2` is a multiple of `13`. If this check passes, `noa_num` is then set to be equal to `(noa_len - 2)/13`.

[4]Protection mechanisms such as ASLR and stack canaries make this significantly more complex in practice.
[5]In practice, the target for buffer overflows is often a specific location in memory

This calculated `noa_num` value is subsequently used to iterate over several arrays in `p2pinfo` and directly access them. While the check for valid length ensures that the calculation of `noa_num` will result in an integer value, it neglects to establish an upper bound on `noa_num`.

Listing 2.1: The vulnerable `noa_len` check in `rtl_p2p_noa_ie` and `rtl_p2p_action_ie`. Taken from [1, 14].

```c
if ((noa_len - 2) % 13 != 0) {
    RT_TRACE(rtlpriv, COMP_INIT, DBG_LOUD,
         "P2P notice of absence: invalid length.%d\n",
         noa_len);
    return;
}
```

This oversight has disastrous implications for unsuspecting users of the vulnerable driver code; any attacker within proximity to initiate a P2P[6] connection can issue specially crafted input to take advantage of the improper bounds check [20].

In the *absolute best case*, the invalid memory access will be caught by the kernel and result in an immediate kernel panic. This effectively constitutes a potential denial of service attack on any system running the vulnerable driver code. As a corollary, an attacker may continuously attempt such a malicious P2P connection to effectively prevent the system from returning to a sane state[7].

In the *worst case*, an attacker may be able to do significantly more damage to the affected system. If the attacker is able to bypass rudimentary protection mechanisms [15] on kernel memory, they may be able to execute arbitrary code in kernelspace or overwrite kernel data structures, leading to possible escalation of privilege, information disclosure, or tampering with the trusted computing base.

2.3 Fixing the Vulnerability

The technical details of fixing a bounds checking vulnerability like the one presented here are fairly straightforward. After identifying the vulnerable variable, a check is put in place to ensure that its value not exceed a specified threshold. Subsequent buffer accesses using that variable are therefore guaranteed to be safe, and the attack vector is therefore neutralized. This technique is known as *runtime bounds checking* [22]. Listing 2.2, lines 9-13, shows the patch applied to the Realtek driver code that implements this simple bounds check.

[6]Peer-to-peer.

[7]Although a knowledgeable system administrator could mitigate this; for example, by `chroot`ing into the system from live media and disabling the vulnerable driver module.

Listing 2.2: The patch applied to fix the `noa_len` check in `rtl_p2p_noa_ie` and `rtl_p2p_action_ie`. Adapted from [1, 14].

```
 1 /* The original check */
 2 if ((noa_len - 2) % 13 != 0) {
 3     RT_TRACE(rtlpriv, COMP_INIT, DBG_LOUD,
 4          "P2P notice of absence: invalid length.%d\n",
 5          noa_len);
 6     return;
 7 }
 8 /* The following branch was added to the above conditional... */
 9 else {
10     noa_num = (noa_len - 2) / 13;
11     if (noa_num > P2P_MAX_NOA_NUM)
12         noa_num = P2P_MAX_NOA_NUM;
13 }
```

While the fix shown in Listing 2.2 is relatively simple, the way that the kernel maintainers have decided to handle the bounds check merits some discussion. In particular, we can see on line 12 of the above code listing that `noa_num` values exceeding the maximum threshold are silently set to that maximum; while this is certainly a valid option, it may lead to unexpected outcomes from the user's perspective. Such an outcome might at least warrant printing a warning to kernel logs. Alternatively, the function could simply return as it does in the first branch of the conditional.

2.4 User Impact and Upgrade Path

Kernelspace buffer overflows can be extremely dangerous, as we have shown in previous sections. Although exploiting the vulnerability we have presented here requires that the vulnerable wifi driver be loaded and that the attacker be within close proximity to the target system[8], this should not detract from the severity of damage that such an exploit may cause. Ultimately, any form of arbitrary code execution in kernelspace, no matter how minute or obscure the attack surface, should be treated with the respect it deserves. Patching such vulnerabilities is of paramount importance to the maintenance of secure systems, particularly in production environments.

Since the Linux kernel is used in a great variety of distributions, the upgrade path to fix this vulnerability will vary to some degree. Generally, it is as simple as downloading a patched version of the kernel (in this case, 4.19.82 LTS or 5.3.6), installing it, and rebooting the system. Unfortunately, distributions tend to handle such upgrades very differently, and some are much better than others at releasing new kernel versions in a timely manner.

[8]In order to initiate a P2P connection with the vulnerable host.

3 FreeBSD: cdrom Driver Privilege Escalation

It was discovered in July of 2019 that a vulnerability in the FreeBSD cdrom driver allowed unprivileged users with read access to a cdrom character device to overwrite arbitrary kernel memory [6, 21, 32]. Specifically, the vulnerability related to the cdrom driver's handling of certain `ioctl` [11] calls on its character device special file. Attackers could make specific `ioctl` calls to copy arbitrary memory into kernelspace, and were able to do this with readonly access into the corresponding special file [32]. This vulnerability affected FreeBSD versions 11.2 to 12.0 [6, 21], although patches for 11.x as well as 12.x versions were made available [30, 31].

3.1 The Vulnerability in Detail

In UNIX-like systems such as FreeBSD, special files are a commonly used interface into various subsystems within the kernel, for example, device drivers. Userland processes may interact with these files via file manipulation system calls in order to read information from, write information to, or otherwise modify parameters of the driver. This relationship is depicted in Figure 3.1. The vulnerability we are describing here was related to how FreeBSD's reference monitor mediated access to the `ioctl` system call on cdrom driver special files [32]. This allowed a user with read access to the device to copy memory into kernelspace, and thus enabled a relatively straightforward escalation of privilege.

The `ioctl` system call is very flexible and by convention is generally used to manipulate parameters on special files [11], such as the character device file exposed by the cdrom driver [32]. Drivers implement their own endpoints for calls like `ioctl`, which can lead to unintended consequences when such calls are handled improperly. The cdrom driver's `ioctl` implementation had a vulnerable branch in its switch statement on user provided parameters; the case `CDIOCREADSUBCHANNEL_SYSSPACE` [30, 31] allowed arbitrary memory to be copied into the kernel's address space via the `bcopy` function, and was erroneously made accessible from userspace for any user with read permissions on the device [32].

In order to exploit the vulnerability, a user in the `operator` group needed read access (which is default for members of this group) to a cdrom device with media present (i.e. a CD-ROM inserted into the drive) [32]. With this configuration, a simple C[9] program could be written to make the appropriate `ioctl` call on the special file.

[9] It need not necessarily be a C program, but C is arguably the easiest language with which to make arbitrary system calls.

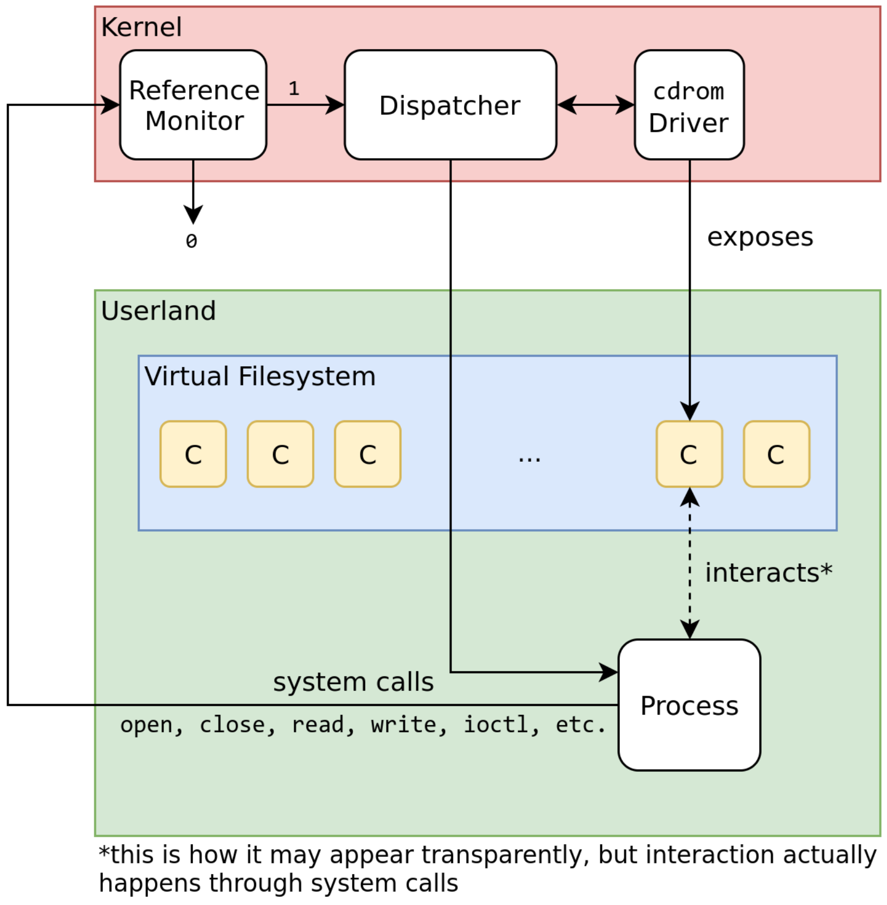
Kernel
Reference Monitor
1
Dispatcher
cdrom Driver
0
exposes
Userland
Virtual Filesystem
C
C
C
...
C
C
interacts*
system calls
open, close, read, write, ioctl, etc.
Process
*this is how it may appear transparently, but interaction actually happens through system calls

Figure 3.1: The relationship between kernel drivers and special files in UNIX-like systems such as FreeBSD. Userland processes are able to interact with drivers by interacting with their character special files in a virtual filesystem. When the system calls required to do so are mediated correctly, this interaction is harmless; but oversights may allow attackers to easily run exploits in kernelspace.

3.2 Fixing the Vulnerability

In order to fix the vulnerability, the FreeBSD development community decided to completely eliminate the erroneous `CDIOCREADSUBCHANNEL_SYSSPACE` in the `ioctl` call. Instead, they elected to use the more generic `CDIOCREADSUBCHANNEL` case, which makes less dangerous assumptions about memory [30, 31]. This fix involved significant refactoring to several parts of both the cdrom driver and FreeBSD's implementation of `ioctl` in the Linux compatibility layer. The changes are summarized as follows [30, 31]:

- removed the vulnerable case for `CDIOCREADSUBCHANNEL_SYSSPACE` in the code for the cdrom driver;
- replaced a check for `nocopyout` in `CDIOCREADSUBCHANNEL` with a simple call to `copyout`;
- added extra checks to the Linux compatibility layer for `ioctl` to ensure safety; and
- changed several function definitions and prototypes to eliminate the user of the `nocopyout` variable.

With this fix in place, specially crafted `ioctl` calls on the cdrom driver would no longer be able to take advantage of the overly permissive interface exposed to userspace. This effectively eliminated the potential privilege escalation exploit.

3.3 User Impact and Upgrade Path

In order to successfully run the privilege escalation exploit on the cdrom driver, an attacker would need access to an account on the target system in the `operator` group, which grants permissions with respect to management of various devices on the system. Additionally, the attacker would need to be able load a specific CR-ROM into a drive, and subsequently open the correct special file and issue a specially crafted `ioctl` call on the open file descriptor. Although the likelihood of this set of circumstances is low, the potential damage caused by such a vulnerability is huge.

Leaving such a dangerous interface exposed to userspace is a gross violation of complete mediation on the part of FreeBSD's reference monitor. Users with read access on a device should *never* be able to overwrite arbitrary kernel memory; such capability completely circumvents the intuitive set of operations that read access should allow. Furthermore, the result of successful execution of this exploit is complete escalation of the attacker's privileges such that they have root access on the target system. This represents a total compromise of the system, since root processes effectively have access to the entire trusted compositing base of the system [13].

Fortunately, the upgrade path to fix this vulnerability is fairly straightforward. Users need only download the applicable patch, apply it, and reboot their system [32]. For convenience, this patch is made available as a source patch for 11.x and 12.x versions [30, 31] as well as a binary patch through FreeBSD's package management system [32].

4 OpenBSD: Unprivileged `ioctl` Calls on Network Drivers

In this section, we will examine *another* vulnerability with the `ioctl` [12] system call, this time under the OpenBSD kernel. In November of 2019, OpenBSD published an errata and an accompanying patch related to a vulnerability in its implementation of `ioctl`; specifically, unprivileged users could modify certain parameters on network interfaces due to insufficient checks within the implementation of the `ioctl` system call [23]. This vulnerability affected OpenBSD versions supporting the vulnerable `ioctl` commands up to and including 6.6 [23].

4.1 The Vulnerability in Detail

We have already discussed the relationship between drivers, special files, and system calls like `ioctl` in Subsection 3.1 and Figure 3.1 on page 7 provides a depiction of how such a relationship manifests in practice. To summarize, drivers expose special files to userland through a virtual filesystem; userland processes interact with these special files by making system calls, which are verified by the reference monitor. Endpoints for these system calls are implemented within the driver itself. As in FreeBSD, Linux, and other UNIX-like operating systems, the OpenBSD `ioctl` [12] system call is primarily used to manipulate the parameters of drivers associated with special files. In order to make this call, the user needs an open file descriptor into the device [12], as well as the permission of the reference monitor [13, 22].

In the case of network interface `ioctl`s in OpenBSD, these calls are made on open file descriptors to network sockets (a special file which serves as an abstraction for a network interface) and generally include an integer representing the desired command as well as (typically) a pointer to an `ifreq`[10] struct containing request parameters [12, 18]. In this way, users can arbitrarily modify a variety of parameters across a variety of network interfaces.

Issues arise when such calls are not correctly mediated by the reference monitor. This was the case for the vulnerable OpenBSD network drivers. A switch statement on the `cmd` argument of `ioctl` calls was missing a staggering 34 cases in the `ifioctl` function, responsible for

[10]Depending on the call, a different struct may be required.

mediating and dispatching calls to `ioctl` on OpenBSD network interfaces [23, 25]. Since these cases would each pass through a call to `suser` to check privileges before falling through, this meant that each of the 34 missing commands was not being correctly mediated [23, 25]; as a result, unprivileged users could provide these commands as arguments to `ioctl` calls in order to be able modify network interface parameters without proper authorization.

In addition to the 34 cases missing in `ifioctl`, the aforementioned call to `suser` to check privileges was missing in two other switch statements related to other `ioctl` calls on network interfaces [23]. This brings the total up to 36 exploitable `ioctl` calls.

Due to the number of exploitable `ioctl` calls, describing each one would venture beyond the scope of this paper. However, for the sake of illustration, we provide one example of a potential interaction resulting from this vulnerability. We will consider the unmediated addition of a new IPv6 address to refer to a specific network interface. First, the malicious user obtains a file descriptor to an IPv6 socket associated with an `inet6` device [10]. They then specify their request in the form of an `ifaliasreq` struct and make an `ioctl` call on the open file descriptor, providing the command `SIOCAIFADDR_IN6` and a pointer to their request struct as a payload. The `ioctl` call succeeds in vulnerable versions of OpenBSD, since this request is not properly mediated.

4.2 Fixing the Vulnerability

In this case, although the scope of the vulnerability was large, fixing it was a fairly straightforward process; OpenBSD developers simply added the 34 missing cases to the switch statement in `ifioctl` and added the two missing checks to `suser`. Now the reference monitor can verify privileges before the `ioctl` calls are dispatched and complete mediation[11] is restored.

4.3 User Impact and Upgrade Path

The user impact of unmediated modification of network interface parameters is potentially quite significant. To reiterate, we have 36 total `ioctl` commands that effectively require no permissions to execute [23]; if an unprivileged user can obtain an open file descriptor to a given socket, they can potentially execute any of these commands on the network interface associated with that socket. Consequentially, malicious actors may tamper with a variety of network interfaces, which may result in unexpected changes in functionality, redirection of legitimate addresses to illegitimate ones, denial of service, and even escalation of privilege through the modification of the network interfaces that TCB network-facing services rely

[11]At least with respect to every possible `ioctl` command on network interfaces.

on. In short, unmediated arbitrary modification of network interfaces by unprivileged users potentially spells doom for any secure system.

As with the other vulnerabilities we have discussed thus far, the upgrade path for users is to patch the kernel, recompile, and reboot the system [23]. OpenBSD provides a straightforward way of doing so through their Anonymous CVS command line interface [8, 24], which provides a mechanism to track *errata*[12] as well as quickly and easily synchronize the local kernel with the upstream source.

5 Windows: Windows Imaging API Remote Code Execution

In October of 2019, Microsoft announced a vulnerability with the Windows Imaging API [5, 7, 17, 19, 26] that allowed for remote code execution through the corruption of memory. This vulnerability affected several versions of windows 10 up to and including version 1903, as well as Windows 7, Windows 8.1, and several editions of Windows Server 2012, 2016, and 2019 [5]. The Windows Imaging API provides an application programming interface for the creation, modification, and deployment of system images, supported by the Windows Kernel [16]. It is generally used in corporate environments for the deployment of uniformly configured system images.

5.1 The Vulnerability in Detail (or Lack Thereof)

Documentation on the vulnerability is sparse, describes the problem in vague, generic terms, and generally provides the same information with little variation [5, 7, 17, 19, 26]. This is primarily due to the nature of Microsoft Windows as a *closed source* operating system, in stark contrast with the open source operating systems[13] that we have examined thus far. We will attempt to further contrast the nature of open and closed source systems with respect to vulnerabilities in Subsection 6.1.

While Microsoft refused to release specific details of the vulnerability, including source code and a specific description of the attack vector, they did reveal that it requires the execution of a specially crafted `.WIM`[14] file. The vulnerability resides in how the Windows Imaging API treats such files with respect to its management of memory [5, 7, 17, 19, 26]; this specially

[12]This is OpenBSD's generic term for known issues in the operating system, including security issues.
[13]GNU/Linux, FreeBSD, and OpenBSD in this paper.
[14]This is the file extension used for Windows system images.

crafted file can cause it to corrupt specific locations in memory, providing an attack vector for code execution.

5.2 Fixing the Vulnerability

Just as in the description of the vulnerability itself, Microsoft again provides *very* little information regarding how it was fixed [17]. What they do reveal is that the fix was related to the treatment of memory by the Imaging API [17]; we can conjecture that this involved the introduction of more checks with respect to input sanitization.

5.3 User Impact and Upgrade Path

Since the successful exploitation of this vulnerability requires the *execution* of a specially crafted file [17], this introduces non-negligible variation in the attack vector presented to attackers depending on whether they are a malicious insider, or an outside actor; as such, we will consider both cases when discussing user impact.

Malicious Insiders. Malicious insiders are the most dangerous of the two options we have presented. Such adversaries may either have direct access to an account on the target system or may be able to gain access to a target account on unmonitored devices. Additionally, disgruntled system administrators could easily replace legitimate system images within some database with malicious ones, designed to execute the exploit at a later point in time, when next used. This type of an attack requires only that the attacker possess the knowledge to craft the malicious `.WIM` file [17].

Outside Actors. The threat model associated with an outside actor with respect to this exploit is significantly weaker when compared to that of the malicious insider. This is due to the simple fact that the execution of the exploit now requires a social engineering component; specifically, attackers need to convince the target user to click on the malicious `.WIM` file [17, 26]. Depending on the technical expertise of the target user, the level of difficulty associated with this extra step may vary significantly. As we have discussed previously, corporate environments would be an ideal target for such an exploit, since they provide the primary use case for the Imaging API in the first place [16]. Depending on the corporation, it may be easier or harder to find vulnerable users susceptible to social engineering attacks. For example, an insurance company might be more vulnerable than a software development firm.

Windows 10 has arguably the most effective upgrade path of any of the systems we have discussed thus far. Rather than requiring a system administrator to manually download and install updates, Windows will automatically download updates and apply them the next time the machine is restarted. A notable exception to this occurs in many corporate environments[15], where system administrators will purposely exert finer grained control over the deployment of updates; in this case, the upgrade path is no more complex than any of the previously described systems.

6 Conclusion

In this paper, we have examined four distinct vulnerabilities in four distinct operating systems, three open source, and one closed source. Each of these vulnerabilities is severe in its own right, and each represents a significant oversight on the part of operating system developers. From the perspective of the consumer, it may be tempting to consider these oversights unacceptable, particularly in the case of proprietary commercial products like Microsoft Windows. However, when considering complex general purpose operating systems, programming errors and gaps in logic become – to a certain extent – inevitable, especially as code bases grow to meet modern requirements and expectations. Perhaps this merits separate discussion with respect to the trade-offs between "trimming the fat" to increase reference monitor verifiability and expanding increasingly bloated feature sets for consumer convenience.

Ultimately, while OS vulnerabilities often represent severe threats to system security, they are in many cases inevitable; they are a side effect of designing complex systems for performing complex tasks in a complex modern world. We can certainly take steps to mitigate such issues, but, as we have seen, there is simply no way to outright prevent the often simple programming errors that plague our modern systems.

6.1 On the Nature of Open and Closed Source Vulnerabilities

As a parting thought, we will conjecture about the differences between open source and closed source operating system vulnerabilities, from the perspective of an author writing a paper like this one. When researching operating system vulnerabilities, one cannot help but notice a distinct difference between both the quantity and quality of resources available between open and closed source operating systems. For example, the Windows vulnerability we discussed in Section 5 offered remarkably little insight regarding the exact nature of the issue and precise steps taken to resolve it – from the perspective of corporations, this makes sense, since they would prefer that their trade secrets remain secret; unfortunately, such corporate practices

[15]As we have discussed, these *are* the most likely target for such an exploit.

are often quite harmful to the end user. How can users trust that a system is secure when its inner workings are completely hidden?

Furthermore, the security benefits of the closed source model are a dubious prospect. One may be tempted to erroneously claim that the limited number of vulnerabilities on systems like Windows provides clear evidence that these systems are more secure. This is fallacious logic, since when discussing the number of vulnerabilities for any given system, what we are actually discussing is the *number of disclosed vulnerabilities*. This is a subset of the *number of known vulnerabilities*, which in turn is a subset of the *number of actual vulnerabilities*.

In contrast, the open source model is far more conducive to open and constructive discussions about operating system security by the group that is most directly affected – the end users. In an open source system, every aspect of the system is subject to public scrutiny and found vulnerabilities are made public domain; in such a system, the sets of disclosed, known, and actual vulnerabilities will be close approximations of one another.

References

[1] L. Abbott. (Oct. 2019). [patch] rtlwifi: Fix potential overflow on p2p code, [Online]. Available: https://lkml.org/lkml/2019/10/16/1226 (visited on 01/20/2020).

[2] Arch Linux. (Nov. 2019). Arch linux security advisory asa-201911-11, [Online]. Available: https://security.archlinux.org/ASA-201911-11 (visited on 01/12/2020).

[3] Canonical. (Oct. 2019). Ubuntu cve tracker cve-2019-17666, [Online]. Available: https://people.canonical.com/~ubuntu-security/cve/2019/CVE-2019-17666.html (visited on 01/13/2020).

[4] Cloudflare. (2020). What is buffer overflow? [Online]. Available: https://www.cloudflare.com/learning/security/threats/buffer-overflow (visited on 01/12/2020).

[5] CVE Details. (Oct. 2019). Vulnerability details: Cve-2019-1311, [Online]. Available: https://www.cvedetails.com/cve/CVE-2019-1311/ (visited on 02/02/2020).

[6] CVE Details. (Jul. 2019). Vulnerability details: Cve-2019-5602, [Online]. Available: https://www.cvedetails.com/cve/CVE-2019-5602/ (visited on 01/31/2020).

[7] CVE Mitre. (Oct. 2019). Cve-2019-1311, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1311 (visited on 02/02/2020).

[8] *Cvs(1) general commands manual cvs(1)*, OpenBSD. [Online]. Available: https://man.openbsd.org/cvs (visited on 02/02/2020).

[9] Debian. (Oct. 2019). Cve-2019-17666, [Online]. Available: https://security-tracker.debian.org/tracker/CVE-2019-17666 (visited on 01/13/2020).

[10] *Inet6(4) device drivers manual inet6(4)*, OpenBSD, May 2019. [Online]. Available: https://man.openbsd.org/OpenBSD-6.6/inet6.4 (visited on 02/02/2020).

[11] *Ioctl(2) bsd system calls manual ioctl(2)*, The FreeBSD Project, Sep. 2013. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=ioctl&sektion=2 (visited on 01/31/2020).

[12] *Ioctl(2) system calls manual ioctl(2)*, OpenBSD, Aug. 2019. [Online]. Available: https://man.openbsd.org/ioctl.2 (visited on 01/31/2020).

[13] T. Jaeger, *Operating system security*. Morgan & Claypool Publishers, 2008.

[14] Linux. (Jan. 2020). Drivers/net/wireless/realtek/rtlwifi/ps.c, [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/wireless/realtek/rtlwifi/ps.c (visited on 01/27/2020).

[15] Linux Kernel Documentation. (Jan. 2020). Kernel self-protection documentation, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-17666 (visited on 01/29/2020).

[16] Microsoft. (2011). Introduction to the imaging apis for windows, [Online]. Available: https://msdn.microsoft.com/en-us/windows/desktop/dd851933.aspx (visited on 02/02/2020).

[17] Microsoft. (Oct. 2019). Cve-2019-1311 | windows imaging api remote code execution vulnerability, [Online]. Available: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-1311 (visited on 02/02/2020).

[18] *Netintro(4) device drivers manual netintro(4)*, OpenBSD, Jul. 2018. [Online]. Available: https://man.openbsd.org/netintro.4 (visited on 02/02/2020).

[19] NVD NIST. (Oct. 2019). Cve-2019-1311 detail, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-1311 (visited on 02/02/2020).

[20] NVD NIST. (Oct. 2019). Cve-2019-17666 detail, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-17666 (visited on 01/12/2020).

[21] NVD NIST. (Jul. 2019). Cve-2019-5602 detail, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-5602 (visited on 01/31/2020).

[22] P. C. van Oorschot, *Computer security and the internet: Tools and jewels*, Sep. 2019. [Online]. Available: https://people.scs.carleton.ca/~paulv/toolsjewels.html (visited on 01/12/2020).

[23] OpenBSD. (Nov. 2019). Openbsd 6.6 errata 006, [Online]. Available: https://ftp.openbsd.org/pub/OpenBSD/patches/6.6/common/006_ifioctl.patch.sig (visited on 02/01/2020).

[24] OpenBSD. (Feb. 2020). Openbsd anonymous cvs, [Online]. Available: https://www.openbsd.org/anoncvs.html (visited on 02/02/2020).

[25] OpenBSD. (Jan. 2020). Sys/net/if.c, [Online]. Available: https://github.com/openbsd/src/blob/master/sys/net/if.c (visited on 02/02/2020).

[26] Rapid7. (Oct. 2019). Microsoft cve-2019-1311: Windows imaging api remote code execution vulnerability, [Online]. Available: https://www.rapid7.com/db/vulnerabilities/msft-cve-2019-1311 (visited on 02/02/2020).

[27] Red Hat. (Oct. 2019). Cve-2019-17666, [Online]. Available: https://access.redhat.com/security/cve/CVE-2019-17666 (visited on 01/12/2020).

[28] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012, ISSN: 1094-9224. DOI: 10.1145/2133375.2133377. [Online]. Available: https://doi.org/10.1145/2133375.2133377 (visited on 01/31/2020).

[29] SUSE Linux. (Oct. 2019). Cve-2019-17666 | suse, [Online]. Available: https://www.suse.com/security/cve/CVE-2019-17666/ (visited on 01/13/2020).

[30] The FreeBSD Project. (Jul. 2019). Cd_ioctl.11.patch, [Online]. Available: https://security.FreeBSD.org/patches/SA-19:11/cd_ioctl.11.patch (visited on 01/31/2020).

[31] The FreeBSD Project. (Jul. 2019). Cd_ioctl.12.patch, [Online]. Available: https://security.freebsd.org/patches/SA-19:11/cd_ioctl.12.patch (visited on 01/31/2020).

[32] The FreeBSD Project. (Jul. 2019). Freebsd-sa-19:11.cd_ioctl, [Online]. Available: https://www.freebsd.org/security/advisories/FreeBSD-SA-19:11.cd_ioctl.asc (visited on 01/31/2020).