NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

INCORPORATING PERISHABILITY AND OBSOLESCENCE INTO CYBERWEAPON SCHEDULING

by

Michael R. Lidestri

June 2022

Thesis Advisor: Neil C. Rowe
Co-Advisor: Wade L. Huntley

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
INCORPORATING PERISHABILITY AND OBSOLESCENCE INTO CYBERWEAPON SCHEDULING

5. FUNDING NUMBERS

6. AUTHOR(S) Michael R. Lidestri

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

As cyberspace operations become further integrated into operational planning for nation-states, planners must understand the implications of perishability and obsolescence when deciding how to use cyberweapons. Obsolescence reflects the risk that a vulnerability will be patched without cyberweapon use, while perishability describes the short lifespan of a cyberweapon once it is used; one creates an incentive to use and the other an incentive to stockpile. This thesis examined operating-system vulnerabilities over four years: we quantified the duration between key events of their life cycles as well as the time to release a patch after disclosure. We performed survival analysis for longevity and post-disclosure patch time using Kaplan-Meier curves, then found that the data fit well to Weibull distributions. We also examined the effects of severity and operating system on the lengths of vulnerability life-cycle phases. Our parametric models enable planners to predict the expected survival time of a cyberweapon's vulnerability, allowing them to determine when to use them, replenish them, and assess windows of opportunity for reuse. This reduces the need to stockpile cyberweapons and creates incentives to use them before the expected survival time. The observed wide variability in longevity values indicates that risk tolerance is important in deciding when to use a cyberweapon.

14. SUBJECT TERMS
cyberweapon, perishability, obsolescence, cyber warfare

15. NUMBER OF PAGES
99

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

INCORPORATING PERISHABILITY AND OBSOLESCENCE INTO CYBERWEAPON SCHEDULING

Michael R. Lidestri
Lieutenant Commander, United States Navy
BSM, Georgia Institute of Technology, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN CYBER SYSTEMS AND OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
June 2022

Approved by: Neil C. Rowe
Advisor

Wade L. Huntley
Co-Advisor

Alex Bordetsky
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As cyberspace operations become further integrated into operational planning for nation-states, planners must understand the implications of perishability and obsolescence when deciding how to use cyberweapons. Obsolescence reflects the risk that a vulnerability will be patched without cyberweapon use, while perishability describes the short lifespan of a cyberweapon once it is used; one creates an incentive to use and the other an incentive to stockpile. This thesis examined operating-system vulnerabilities over four years: we quantified the duration between key events of their life cycles as well as the time to release a patch after disclosure. We performed survival analysis for longevity and post-disclosure patch time using Kaplan-Meier curves, then found that the data fit well to Weibull distributions. We also examined the effects of severity and operating system on the lengths of vulnerability life-cycle phases. Our parametric models enable planners to predict the expected survival time of a cyberweapon's vulnerability, allowing them to determine when to use them, replenish them, and assess windows of opportunity for reuse. This reduces the need to stockpile cyberweapons and creates incentives to use them before the expected survival time. The observed wide variability in longevity values indicates that risk tolerance is important in deciding when to use a cyberweapon.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

CDF	cumulative distribution function
CISA	Cybersecurity and Infrastructure Security Agency
CPE	common product enumerator
CSV	comma-separated value
CVE	common vulnerabilities and exposures
CVSS	common vulnerability scoring system
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
NIST	National Institute of Standards and Technology
NVD	National Vulnerabilities Database
OSVDB	Open-Source Vulnerabilities Database
TCP	Transfer Control Protocol
URL	Uniform Resource Locator
XML	Extended Markup Language
WINE	Worldwide Intelligence Network Environment

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Neil Rowe, for his assistance and guidance throughout the thesis production process. I would also like to thank my co-advisor, Dr. Wade Huntley, for his help in selecting a thesis topic and guidance he provided while writing my thesis. I would like to thank all my classmates and fellow military officers, without whom I could not have succeeded. Lastly, I would like to thank my wife, Yukiko, for supporting me during my tenure at the Naval Postgraduate School, and for carrying our daughter, who is due in August.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Exploits are malicious code that leverages software vulnerabilities to insert a payload onto a target information system (Herr & Rosenzweig, 2014). Vulnerabilities are software bugs (flaws) that allow a malicious actor to attack the confidentiality, integrity, or availability of an information system (Microsoft, 2013). Bugs are prevalent in many software programs; an estimated 20 occur for every 1,000 lines of code (Dacey, 2003). For reference, the Linux kernel contains 27.8 million lines of code (Anderson, 2020). Vulnerabilities persist in software and remain exploitable unless they are removed by corrective software updates called patches (Sarabi et al., 2017). Vulnerabilities that remain unpatched because the vendor is unaware of their existence are called zero-day vulnerabilities and the corresponding exploits are zero-day exploits (Libicki et al., 2015). A cyberweapon is an exploit that is used in a conflict to either inflict physical damage or to sabotage or damage an information system; cyberweapons allow actors to execute cyberspace attacks (cyberattacks) against a target using information systems (Mele, 2014).

Because of the low risk and high potential reward of carrying out successful cyberattacks, actors of varying sizes and ideologies do so to further their own interests (U.S. Cyber Command, 2018). While smaller actors may participate in cybercrime, theft of intellectual property, or online activism, some state actors have poured significant resources into their cyber capabilities and can operate more sophisticated campaigns. In 2012, the Shamoon virus deleted data from the hard drives of over 30,000 Windows machines on Saudi Aramco's network; it was speculated that Iranian actors were responsible for the attack (Bronk & Tikk-Ringas, 2013). The Iranians themselves were victims of the earlier Stuxnet worm, which used multiple Microsoft Windows zero-day exploits and stolen digital certificates to modify the function of Siemens programmable logic controllers (Langner, 2013). This attack used two vectors, overpressure and rotor speed, to attempt to induce physical damage to Iranian centrifuges. In 2015, a cyberattack on three Ukrainian electricity distribution companies resulted in a loss of power for 225,000 people, which the Ukrainian government blamed on Russia (Lee et al., 2016). Established powers like the United States have also integrated cyber operations into joint

military planning: In 2016, U.S. Cyber Command established Joint Task Force ARES to inhibit Islamic State cyber activities, supporting Operation Inherent Resolve (Martelle, 2018).

Cyberspace operations are defined in U.S. doctrine () as "the employment of cyberspace capabilities where the primary purpose is to achieve objectives in and through cyberspace" (U.S. Joint Chiefs of Staff, 2018, p. II-1). These capabilities are executed across three layers of the information domain: physical, logical, and cyber-persona. The physical layer includes physical network components and IT infrastructure that are represented by their geographic location. The logical layer describes an abstraction consisting of components that belong to the same network; physical location may vary. The cyber-persona layer represents the many digital identities that people maintain based on the roles they fulfill or their online behaviors. Cyber operations can be either offensive or defensive, and they involve actions to attack, defend, exploit, or secure networks or information systems. Cyberspace attack actions are executed to create effects in cyberspace or in the physical domain.

Unlike other domains, the dynamic nature of cyberspace means that the opportunity to use or reuse cyberweapons may be limited, complicating cyber operations planning (Smeets, 2018). Cyber exploits are often considered to have a one-time capability with limited efficacy: After an exploit is deployed and subsequently detected, a patch (fix) will normally be quickly developed. Once this occurs, the underlying vulnerability is no longer available, and the exploit is ineffective. Therefore, cyberweapons that exploit software vulnerabilities are sometimes described as being transitory or perishable (Huntley, 2016; Smeets, 2018). According to Smeets, this concept of transitoriness differentiates cyberweapons from physical weapons, and because of this, planners must consider how and when to deploy cyberweapons and which targets should be prioritized.

Even when it is not used, a cyberweapon may lose effectiveness over time, a trait called obsolescence that suggests that vulnerabilities have a finite life cycle (Huntley, 2016). Obsolescence can occur when a vulnerability is disclosed by a third party or when code is updated that eliminates the vulnerability during software development (Ablon & Bogart, 2017). This creates an incentive to use a cyberweapon before it will no longer

provide desired effects (Huntley, 2016). While the window of opportunity may not disappear completely following disclosure of a vulnerability, once a patch is released, a rapid decline in the number of exploitable systems occurs (Frei et al., 2009), quickly diminishing the effectiveness of the exploit.

A thorough understanding of the factors affecting perishability and obsolescence is necessary to predict how long a cyberweapon is likely to remain effective. This enables offensive planners to make decisions about which exploits to keep in inventory, when to deploy them, and their period of for reuse. Because we can measure the duration between each of the events associated with the software vulnerability life cycle, we can model the probability of survival from each event using survival analysis techniques, which will allow us to predict a cyberweapon's period of usefulness (Izmiguzel, 2021).

Previous research has studied the software vulnerability life cycle (Arbaugh et al., 2000), the phases through which a vulnerability can progress. While later research has refined the life cycle and quantified the length of some phases, most studies have focused on the relationship between when vulnerabilities are disclosed and the development of patches or exploits (Frei et al., 2006; Shazad et al., 2020; Arora et al., 2010). Other studies (Frei et al., 2006; Ablon & Bogart, 2017) have examined the timeframe from discovery to disclosure. Several studies have also analyzed other factors that may affect the life cycle, including the effect of vulnerability severity on the time required for vendors to release patches, disparities in open versus closed-source vulnerability disclosure and patching (Shazad et al., 2020), and the effect of code familiarity and reuse in the discovery of vulnerabilities in newly released software (Clark et al., 2010).

However, there has not been much effort to determine the total lifespan of a vulnerability from when it is created to when it is disclosed and patched. This is significant considering the extent of code reuse and legacy software (old software still used) today. It is also relevant to cyber operations planning because the remaining useful life of an exploit could then be estimated, informing risk versus gain calculations for deploying it.

This thesis examines operating system vulnerabilities from the National Vulnerabilities Database (NVD) from 2018–2021. Because the Open-Source

Vulnerabilities Database (OSVDB) used in previous studies no longer exists (Kovacs, 2016), different data sources were used, including vendor security bulletins, exploit databases, the NVD, and other public information pertaining to the creation, disclosure, and patching of vulnerabilities and the development of exploits. Chapter II describes previous research studying concepts of cyberweapons, the software vulnerability life cycle, and factors affecting the length of phases of the life cycle. Chapter III explains perishability and obsolescence, and their relationship to the software vulnerability life cycle, and introduces the datasets used in our analysis. Chapter IV describes the methods used for collecting data. Chapter V contains results and statistical analysis, while Chapter VI discusses the strategic implications of our results, and how they apply to planning for cyberspace operations.

II. RELATED RESEARCH

A. OVERVIEW

The purpose of cyberwarfare is to create desired effects, both in cyberspace and physically (U.S. Joint Chiefs of Staff, 2018). Due to the low cost of entry, many actors can do cyber operations. While the level of sophistication and complexity may vary between cyberattacks, there are several common steps that must be taken to execute an attack. According to a cyber kill chain model developed by Lockheed Martin (Hutchins et al., 2011), the first steps in this process start with reconnaissance of the target to identify vulnerabilities and potential methods of entry. Once vulnerabilities have been identified, a suitable malware program must be obtained or developed to exploit the vulnerability.

B. DEFINING CYBERWEAPONS

Malware can be classified by the target operating system, target device, dependencies specific to that malware, how it propagates itself between devices, or the method by which malware is introduced to a target system (physical media, email, chat, URLs, file sharing services, or software vulnerabilities) (Elisan, 2015). The classes of malware derived from these characteristics include infectors, network worms, trojan horses, backdoors, remote-access trojans, information stealers, ransomware, scareware, fakeware, and greyware. Malicious programs may fall in multiple categories, and they may be further differentiated by their infection vectors and dependencies. While software vulnerabilities exploit bugs or flaws in software code, other infection vectors involve using otherwise legitimate software for malicious purposes. Malware dependencies are conditions on the target system for malware to execute, such as the operating system, virtualization, system settings, or software programs present such as a web browser. Other dependencies include timing-based or event-based conditions that trigger the execution of malicious software. Users can also be dependencies because some malware may require certain levels of privilege or access to execute. Still others, such as information stealers, may depend on specific files stored on the target system.

To define cyberweapons, Liles and Poremski (2015) combined the taxonomies for malware and weaponry to guide risk assessment and incident response. The increasing complexity and adaptability of malware has decreased the usefulness of behavioral characteristics as classification criteria. After behavior, each class was further classified based on "weaponess," which represents the ability of a class to cause damage or harm; while this varied from one class to the next, it was found that all classes of malware include some deception to escape detection and survive.

Defining cyberweapons from a legal perspective, Mele (2014) used the characteristics of context, purpose, and means (or tools). Context in cyberwarfare is conflict among actors where an advantage is obtained using information systems. This distinguishes cyber warfare from activities like cybercrime and clarifies the intent for dual-use software or devices. The purpose requires that malware inflict damage to physical entities, or damage or deny use of the target's information systems. Lastly, means requires that information systems or networks be the primary mode of attack. Putting these three elements together, Mele defines a cyberweapon as

> a part of equipment, a device, or any set of computer instructions, used in a conflict among actors both National and non-National, with the purpose of causing (directly or otherwise) physical damage to objects or people, or of sabotaging and/or damaging in a direct way the information systems of a sensitive target of the attacked subject. (p.61)

C. THE SOFTWARE VULNERABILITY LIFE CYCLE

Software vulnerabilities are common; it is estimated that 20 bugs occur per thousand lines of software code (Dacey, 2003). Not every bug is a vulnerability and not every vulnerability can be exploited. Several studies have focused on how software vulnerabilities are created, disclosed, exploited, and patched.

Arbaugh et al. (2000) first proposed a life cycle model to describe the phases of a vulnerability in its lifetime, starting with birth and continuing through to discovery, disclosure, correction, publicity, scripting, and death. Vulnerabilities normally occur during software development. After software release, they are eventually discovered by motivated actors, whose goals may be either benevolent (white hats) or malicious (black

hats). Correction begins when the vulnerability is disclosed and its existence is revealed to a larger audience that includes the vendor, and patch development starts. Eventually, the public becomes aware of the vulnerability (publicity). As more people learn of the vulnerability, knowledgeable hackers develop exploits, and write and distribute executable scripts or other tools to use them. This allows actors with little expertise to use the exploits, increasing the number of malicious actors and therefore the number of attacks on unpatched systems. Finally, death of the vulnerability occurs when all affected systems have been patched or are removed from service due to replacement or retirement.

Frei et al. (2006) describe a similar life cycle model with a timeline for discovery, disclosure, exploit availability, and patch release, which was developed for over 14,000 vulnerabilities in the NVD for 1996–2006. The period between discovery (when a vulnerability is documented as being found) and disclosure (when a vulnerability has been analyzed and published publicly by trusted sources) is the zero-day state when the risk is greatest. While most vulnerabilities are disclosed quickly after discovery, since 2003 more than 20% of vulnerabilities were discovered at least 20 days before they were disclosed. When observing the cumulative distribution of the time required to develop an exploit relative to disclosure ($t_{exploit}$ - $t_{disclosure}$), nearly 70% of exploit reports since 2001 were followed on the same day by vulnerability disclosures, and 95% of exploits were released within a month following disclosure. According to Frei et al., the large share of published exploits on the date of disclosure suggests that many vendors respond quickly to published exploits. Alternatively, they note that the 95% share of exploits published within a month following disclosure could mean that black hat actors are becoming more adept at reverse engineering.

The cumulative distribution of the time required to develop a patch relative to disclosure ($t_{patch} - t_{disclosure}$) showed that since 2001 most patches were initially available on the date of disclosure (Frei et al., 2006). A comparison of the cumulative distributions for patch and exploit availability relative to the date of disclosure showed that the percentage of exploits that were available out to 300 days from disclosure exceeded the share of patches that were released, indicating that there is some post-disclosure risk of a vulnerability being exploited before a vendor can release a patch.

Shazad et al. (2020) expanded on Frei's work, examining trends in exploit and patching behavior for 56,000 vulnerabilities from 1988–2013. Data was combined from the NVD, OSVDB, and the Frei study; the eight vendors with the most vulnerabilities were selected for analysis. As with previous work, the study focused on the dates for both exploit and patch availability relative to the date of disclosure ($t_{exploit} - t_{disclosure}$ and t_{patch} - $t_{disclosure}$). Most exploits were first published on the disclosure date for the underlying vulnerability, and since 2004 that share has increased. This was attributed to the actions of security experts and other benign actors; the number of exploits available before disclosure appeared to be decreasing over time. For open-source vendors, more exploits were developed prior to the date of disclosure compared to closed-source vendors; Shazad et al. speculated that this is probably due to malicious actors taking advantage of the available source code. However, they observed that Microsoft and Apple products were more likely to have exploits published on or before disclosure, reflecting the ubiquity of the products and the divergent motivations for actors to develop exploits and disclose them. Across product lines, Windows and Firefox were exploited faster than their peers, as each showed a higher proportion of exploits released before disclosure.

Regarding patching, closed-source vendors are faster at releasing patches (Shazad et al., 2020); over 70% of vulnerabilities were patched on or before the disclosure date, suggesting that for-profit companies provide more responsive product support. For vulnerabilities where both exploit and patch dates were available, a comparison of the two ($t_{exploit} - t_{patch}$) determined that 46% of exploits were released before the corresponding patch. This compares to 31.7% of vulnerabilities where patches preceded exploits, indicating that hackers were more successful at exploiting vulnerabilities before vendors patched them. However, this trend reversed in later years; in 2011, 77 percent of patches were released before the corresponding exploits. Across products, Microsoft and Sun Microsystems, respectively, patched Windows and Solaris vulnerabilities more quickly than hackers released exploits. Across vendors, only Oracle and Sun Microsystems patched more vulnerabilities before they were exploited than vice versa.

Shazad et al. (2020) also found vulnerabilities in common between products of the same line (e.g., Windows) while finding comparatively few between vendors, such as

Linux products from Ubuntu, Red Hat, and Fedora. This suggests that vendors re-use much of their code in different products. By contrast, there were very few code similarities between vendors, even those based on the Linux kernel. Therefore, diversity across vendors can reduce the effects of specific vulnerabilities, while using multiple operating systems from a single vendor does not.

1. Discovery

Clark et al. (2010) observed a "honeymoon effect" where vulnerability discovery was delayed following product release, but intervals between successive vulnerability discoveries became smaller each time. By measuring version release and disclosure data for 30,000 vulnerabilities covering 700 software applications, they found that in 62% of cases, the interval from software release to the first vulnerability disclosure (p_0) was longer than the interval separating disclosure of the second successive vulnerability (p_1). The median honeymoon period was 110 days, while the Honeymoon ratio (p_0:p_1) was 1.54 overall and 1.8 for major releases, reflecting smaller intervals for discovery of the second vulnerability.

These results were unexpected; there should be more bugs in newer software, the number of bugs should diminish over time as they are patched, and the remaining bugs should be harder to find (Clark et al., 2010). Surprisingly, 77% of first-discovered vulnerabilities were in legacy code; they were present but undiscovered in earlier software versions, suggesting that many vulnerabilities have significant longevity and persist through multiple software versions. Ultimately, the honeymoon effect was attributed to hackers' degree of familiarity with new software; as the software progresses through its life cycle, hackers become more familiar with its code and thus take less time to discover new vulnerabilities.

2. Disclosure

Arora et al. (2010) observed that vendors will patch a vulnerability more quickly once it has been disclosed. When this interval between vendor awareness of the vulnerability and patch release was measured for 420 vulnerabilities from the National Vulnerabilities Database, immediate disclosure resulted in significantly reduced delays in

patch releases (an average of 28 days versus 65 days for delayed disclosure). This was supported by subsequent statistical hazard modeling indicating that the immediate probability that a patch would be released was 2.5 times higher after disclosure. However, while immediate disclosure raises public awareness and may motivate the vendor to release a patch quickly to maintain its reputation and sales, it also allows malicious actors more time to create and deploy exploits before a patch is available. Delayed disclosure allows the vendor more time to develop a patch before increased exploitation, but vendors do not always act promptly to patch vulnerabilities before they are disclosed, which leaves them open for discovery by others. Unsurprisingly, more severe vulnerabilities had an increased rate of patching.

3. Patch Adoption

Some studies have examined how much users contribute to patching delays. Frei et al. (2009) identified patch update mechanisms, specifically auto-update, as a major factor in patching delays for four Internet browsers (Mozilla Firefox, Microsoft Internet Explorer (IE), Opera, and Apple Safari). Using samples from Google web server logs, user-agent fields in HTTP requests were parsed to extract browser and version data which were compared to version-release dates. Across all browsers, major updates (e.g., IE6 to IE7) were adopted more slowly except when released as an automatic upgrade or bundled with new software. Minor version releases for Firefox and Opera showed a high initial rate of adoption, with a rapid increase in the share of users running the new version, before slowing down. Firefox users, who could often install minor updates with only a single click, adopted patches more quickly; new versions reached 50% share in approximately three days. Opera users, who needed to manually download and install updates, required eleven days to reach majority share. However, because minor version updates are relatively frequent, the share of users operating the most up-to-date version never exceeded 80% for Firefox and 40% for Opera.

Sarabi et al. (2017) studied update delays due to user negligence and patch-update mechanisms. Patch releases were tracked for 1,822 vulnerabilities affecting four products (Google Chrome, Mozilla Firefox, Mozilla Thunderbird, and Adobe Flash Player);

adoption was observed for over 400,000 users using the Worldwide Intelligence Network Environment (WINE) data. Predictably, longer patch delays resulted in more time spent in a vulnerable state. Across all products, an average user was in a vulnerable state more than half of the time, mostly due to user delay in applying updates, which accounted for greater than half of days in a vulnerable state for all products except Chrome (47.9%). However, users less often had long patch delays when the update was automated; the shorter patch delays with Chrome were attributed to the browser's automatic updates. In contrast, Flash users had long patch delays because it required mostly manual downloads and installation.

D. PERISHABILITY AND OBSOLESCENCE

The software vulnerability life cycle assumes that most software vulnerabilities have a finite lifespan, after which patches are released, and the vulnerability becomes less prevalent. Smeets (2018) defines a trait of cyberweapons called transitoriness that describes how they are made less effective over time once the underlying vulnerability is recognized by or disclosed to the vendor. While this loss of effectiveness occurs in both conventional weapons and cyberweapons as adversaries develop countermeasures, the period in which this occurs is much shorter for cyberweapons: a patch can be deployed quickly. A unique feature also defines cyberweapon countermeasures: Once a vendor releases a patch, it is available to all users of the software. Therefore, using a cyberweapon may result in its ineffectiveness against future targets, especially when it causes highly visible damage. This discourages repeated use of a cyberweapon, since it will remain more useful if it is not known. Ultimately, actors must understand the tradeoff between present effects and future use, and they must develop plans that consider the quantity and quality of targets for present and future attack.

Huntley (2016) considers the strategic effects of perishability and obsolescence as one factor in the balance between offensive and defensive advantage in cyberspace. Perishability refers to the prospect that cyberweapon use will reveal the existence of the underlying vulnerability and it will be patched, rendering the cyberweapon unavailable for future use. This creates an incentive to store exploits and creates a higher threshold for conflict in cyberspace. Obsolescence, which refers to the possibility that someone else may

discover and patch the same vulnerability (potentially without an adversary's awareness), results in an incentive to use a cyberweapon, acting in opposition to perishability. Huntley describes the consequences of the interplay between these two factors; it may foster uncertainty regarding whether a cyberweapon will work, from both an offensive and defensive perspective. This uncertainty results in worst-case thinking that transitions into a cyber arms race.

Though cyberweapon use may compromise its use against future targets, it is possible to increase the window of opportunity for reuse (period of successful reuse). Hall (2017) describes several factors that influence the window of opportunity. Some relate to characteristics of the target: Negligent system administrators may not promptly patch their systems or may turn off the auto-update, leaving systems vulnerable to attack. The EternalBlue, WannaCry, and Petya exploits were all derived from a vulnerability (MS 17–010) that permits remote code execution, despite the existence of a patch to correct the vulnerability (Cybersecurity and Infrastructure Security Agency [CISA], 2018). According to Hall, other target characteristics that increase the window of attack opportunity are outdated signatures in antivirus software, vendor delays in releasing patches, and user susceptibility to social engineering. Hall describes how actors may also use persistence and detection evasion to increase exploit survivability. Persistence requires establishing an enduring presence on the host system or network and can be achieved by memory-resident attacks, executable injection, or privileged access through rootkits. Detection evasion can be accomplished through packing (Kang et al., 2007), encryption, or with sophisticated malware that is polymorphic or metamorphic and is therefore able to mutate either its decryption stubs or its body (You & Yim, 2010). Hall further states that evasion can also be aided by inserting exploits at various parts of the supply chain, effectively building the exploit into the product, and making it hard to remove.

E. CONSIDERATIONS FOR CYBER OPERATIONS PLANNING

Ablon and Bogart (2017) examined the cost and benefits of stockpiling zero-day exploits versus disclosing them to avoid compromise of friendly networks. They measured the life expectancy and collision rate (the rediscovery rate) for over 200 zero-day exploits

developed or obtained from 2002–2016. The exploit data was collected from an anonymous research group referred to as BUSBY that has relationships with nation-state actors; the dataset was therefore assumed to be representative of friendly nation actors and their stockpile, and it could also be used to infer a baseline for adversary stockpiles. They compared the data about the exploits and their underlying vulnerabilities against what was in the public domain. Using disclosure and patch data from public sources, as well as exploit data from the developers themselves, exploits were classified as either "living" (undisclosed) or "dead" (public awareness; it was either disclosed or patched), and the life expectancy, defined as the length of time from birth (initial discovery) to death (disclosure or patching), was determined. 40.1% of the vulnerabilities in the sample were found to be dead, while 38.2% were still undisclosed, including 6.3% that would never be patched because the software was no longer supported. The remainder were either eliminated by code refactor (10.1%), where updated software removed the vulnerability without disclosure, or the status was uncertain (11.6%). Using the subset of living and dead exploits for which discovery information was available, the average life expectancy of an exploit was estimated as 6.9 years, although this decreased to 1.4 years when the vulnerability was obtained from an external source. The bottom quartile of exploits was discovered and patched within 1.5 years, while the top quartile survived for at least 9.5 years. This contrasts with the relatively brief time to develop an exploit; 71% were developed in less than 31 days.

While the life expectancy of exploits was long, the collision rate (the rate at which discovered vulnerabilities are then discovered by others) was relatively small: The median rate was about 5.7% for a one-year period, meaning that only 5.7% of exploits would be re-discovered in a year (Ablon & Bogart, 2017). For a 90-day period, the median rate was 0.87%. Because the collision rate is low, an adversary is unlikely to discover the vulnerability and exploit it. Therefore, there is less risk to friendly networks, reducing the pressure for them to disclose the vulnerability. However, Ablon and Bogart also observed that the longevity and low collision rate suggest that there may be little need to stockpile zero-day exploits, although it may be wise to hold a few in inventory for redundancy.

Because of the transitory nature of cyberweapons, Hall (2017) argues that cyber operations planning should be like planning for aircraft combat survivability, where probabilistic assessment of mission success or failure should be considered. If such an assessment determines that the exploit is likely to succeed and remain undetected, the weapon can be reused; however, if detection is likely, measures can be taken to increase survivability. If the exploit is detected, it can be reverse-engineered to determine what vulnerabilities are exploited and how its payload functions so that a patch can be developed. Several of the methods mentioned earlier can make this process more difficult, increasing the opportunity for re-use, and could affect the determination about whether to stockpile a weapon or use it.

Smeets (2018) recommends prioritization of targets for cyberweapons based on the exploit's current state in the software vulnerability life cycle. High-value targets should be attacked during the awareness delay, when only the exploit developer is aware of the vulnerability, to exploit the lack of countermeasures and maximize the odds of success. Once the vulnerability has been disclosed, but the vendor has not yet developed a patch for it (the patching delay), the attacker need not be as selective with targets; the effectiveness of the exploit will soon start to diminish. This period may be somewhat chaotic as other actors create and deploy their own exploits. Lastly, once a patch is released in the "adaptation delay," targets are unpatched systems whose number gradually diminishes.

III. ASSESSING PERISHABILITY AND OBSOLESCENCE IN THE SOFTWARE VULNERABILITY LIFE CYCLE

A. PERISHABILITY AND OBSOLESCENCE

Cyber operations models, such as the Lockheed Martin cyber kill chain (Hutchins et al., 2011), generally begin by conducting reconnaissance of a prospective target. The objective is to get information about the target to identify potential vulnerabilities that can be used to develop and deploy exploits. However, a vulnerability that exists now may get patched in the future. The transitory nature of cyberweapons coupled with the easy availability of software patches means that cyberweapons lose effectiveness much faster than conventional weapons (Smeets, 2018). If a cyberweapon is used and its existence is discovered, such use may stimulate patch development and render it ineffective for future use; this is called "perishability" (Huntley, 2016). The software vulnerability life cycle also suggests that vulnerabilities have a finite lifespan (Arbaugh et al., 2000) and will eventually become obsolete regardless of whether they are used (Huntley, 2016).

The concepts of perishability and obsolescence create opposing incentives for exploit use; perishability provides an incentive to avoid using an exploit because use would quickly result in the vulnerability being patched, while obsolescence provides an incentive to use an exploit because it may eventually be disabled (Huntley, 2016). In research of 200 exploits, Ablon and Bogart (2017) found that the rate of rediscovery for exploits in their dataset was 5.76% annually; while this rate is relatively low, some risk is incurred when storing an exploit for an extended period. Another 10.1% of the exploits studied by Ablon and Bogart were eliminated through "code churn," regular updates to software code that removed the vulnerability without formal disclosure.

Once an exploit has been disclosed, the window of opportunity to use it begins to close, but it usually does not close completely (Smeets, 2018). Providing patches alone does not prevent an exploit from working because users do not always download and install them quickly enough, leaving their systems vulnerable to exploitation (Sarabi et al., 2017). For software programs, a high rate of patching occurs when patches are released (Frei et al., 2009). After the initial burst, patching continues but the percentage of users that are patched

never reaches 100 percent. Therefore, exploits can still maintain some effectiveness even after patches are released. Forty-two percent of vulnerabilities were observed to be exploited after the vulnerability had been disclosed and a patch was released; sometimes several years later (Metrick et al., 2020). This often happens with legacy systems use, which is especially common in both government systems and industrial control systems (Harris, 2019; Johnson, 2018). As of January 2021, an estimated 20% of PCs were still running Windows 7 (Warren, 2021), for which Microsoft had ceased support in 2020, prompting a warning (Federal Bureau of Investigation, 2020).

The software-vulnerability life cycle describes events that may occur during a vulnerability's existence (Frei et al., 2010), as shown in Figure 1. Smeets (2018) identifies three delays in the cycle that affect the transitoriness of an exploit: awareness delay, patching delay, and adaptation delay. These delays correspond to periods in a vulnerability's life cycle that incur gradually increasing levels of risk to the attacker that an exploit will not achieve desired effects due to patching and other countermeasures. Most research of the software-vulnerability life cycle has focused on the length of these periods and their relationship to the time of exploit availability, usually beginning with the disclosure or discovery of the vulnerability (Frei et al., 2006; Frei et al., 2009; Ablon & Bogart, 2017; Shazad et al., 2020).

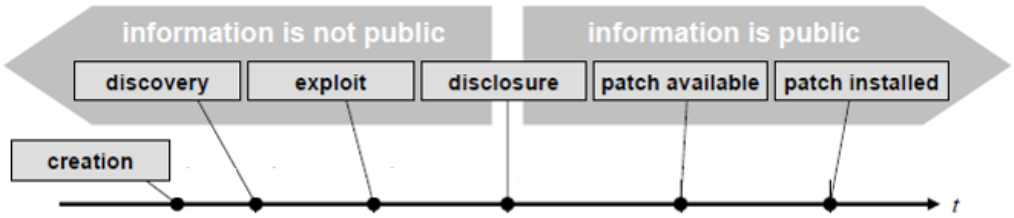

Figure 1. The software vulnerability life cycle. The order of events may vary and some may not occur at all. Adapted from Frei et al. (2010).

While disclosure and discovery are useful milestones to assess the timeliness of subsequent events such as patch release, they provide an incomplete model of the life cycle of a vulnerability because they do not consider when it was created. This information can

often be estimated using product or version release data for the software (Clark et al., 2010). Using this information, we could assess the probability of disclosure based on the age of a vulnerability, providing greater understanding of its obsolescence. This would factor into the risk versus gain considerations that cyberweapons planners must consider when matching a weapon to a target. Understanding the perishability of an exploit will also enable planners to determine the timespan for weapon reuse to prioritize targets.

B. DATA COLLECTION REQUIREMENTS

To model the lifespan of a vulnerability, we must quantify the duration from the creation of a vulnerability to its disclosure and patch release. This requires determining when vulnerabilities were created, perhaps in a previous software version; many operating systems often reuse code to save time and money when creating new products. However, this can also result in legacy vulnerabilities remaining in later software versions. Clark et al. (2010) found that 61% of vulnerabilities attributed to Windows Vista had carried over from previous Windows versions; 40% originated in Windows 2000, suggesting that vulnerabilities can persist undiscovered for long periods. Overall, 77% of the vulnerabilities in their study were also present in earlier software versions.

To determine the lifespan of vulnerabilities, we needed to get the creation date of the earliest affected software version, as well as the disclosure and patch dates. We also needed data of the severity of vulnerabilities, and whether cyberweapon use affects the patch time. To measure cyberweapon use at scale, the thesis analysis assumed that published exploits are like zero-day exploits that have been deployed during cyber operations but were discovered and reverse-engineered by cybersecurity professionals. Therefore, by tracing exploits to their associated vulnerabilities, we could examine differences in patch development between those vulnerabilities that have a corresponding known exploit and those that do not. This required collecting data about published exploits for the vulnerabilities in a dataset.

C. DEFINITIONS, DATA SOURCES, AND ASSUMPTIONS

Following the software-vulnerability life cycle model, data was collected for four dates when possible: when a vulnerability was created ($t_{creation}$), when it was publicly disclosed ($t_{disclosure}$), when a patch was released by the vendor (t_{patch}), and when an exploit

was available for that vulnerability ($t_{exploit}$) (Frei et al., 2010). The discovery date, $t_{discovery}$, was unavailable for all the databases used in the research for this thesis. Data from all sources needed to use CVE ("common vulnerabilities and exposures") identifiers to distinguish vulnerabilities. We extracted the following parameters:

- $t_{creation}$ - the earliest date at which software containing the vulnerability was available for public release
- $t_{disclosure}$ - the date at which information about the vulnerability was made public
- t_{patch} - the date on which a patch was released to address a vulnerability
- $t_{exploit}$ - the date on which when a functional exploit was created

Our primary source for CVE identifiers was the NVD (https://nvd.nist.gov), from which we obtained 68,667 vulnerabilities for the years 2018–2021. This dataset spanned 8,740 unique vendors and over 19,000 software product lines. Our analysis required data from individual vendors as well as manual data collection of some software information. To aid this, we limited data collection to vulnerabilities for the operating-system product lines of Windows, Apple (iOS, tvOS, watchOS, and MAC OS), Android, Google Chrome, and several Linux products (Red Hat, Fedora, Debian, Ubuntu, and the Linux Kernel). This still yielded a dataset of 10,912 vulnerabilities with which to work.

For $t_{creation}$ dates, we used the public release date for the earliest software version in the NVD containing the vulnerability. We did not include dates for beta releases (pre-releases) because they are part of software development. For $t_{disclosure}$ dates, we used the date when the vulnerability was published to the NVD, a public database. For t_{patch} dates, we used the vendor's update release dates (Frei et al., 2006). Exploit data was collected from the Exploit Database (https://www.exploit-db.com); $t_{exploit}$ dates were determined to be the date exploits were published (Nappa et al., 2015). If more than one date existed for a specific CVE identification number, such as when more than one patch was created for vulnerabilities affecting multiple products, we used the earliest date.

To assess perishability, we measured the time required following cyberweapon use for the exploit to be disclosed and patched. This period is the window of opportunity for reuse (Hall, 2017). Because the Exploit Database (Offensive Security, 2022) also publishes exploit code, published dates for $t_{exploit}$ were assumed equivalent to deployment once it has been discovered by cyber incident response personnel and reverse-engineered. Notably, this does not include delays in discovery and reverse engineering; many techniques can extend this timeframe and the window of opportunity to reuse an exploit against subsequent targets (Hall, 2017).

D. DATABASES

1. National Vulnerabilities Database

The NVD is a repository of reported vulnerabilities that is maintained by NIST (Nat. Institute of Standards and Technology [NIST], 2022). Stored vulnerabilities are indexed by their CVE IDs, which are maintained by the MITRE Corporation and assigned by MITRE and an international group of vendors and research personnel. The first four numbers in the CVE ID give the year in which it was issued, and the remaining digits give a serial number for the vulnerability. For example, the first CVE ID assigned in 2018 was CVE-2018-0001.

Vulnerabilities are assigned by NIST a CVSS (common vulnerability scoring system) score measuring their severity using factors of the attack vector, complexity, level of privilege required, scope, and level of required user interaction (NIST, 2022). CVSS overall scores range from 0–10, which is used to determine the severity: critical, high, medium, or low. Vulnerabilities are also assigned weakness enumerators, which place the vulnerability in a family with similar attributes such as memory-buffer errors or exposure of sensitive information. Each CVE ID also has CPEs ("common product enumerators") which identify the products that are vulnerable to that CVE. The format for the fields is specified by (NIST, 2011), but at a minimum, a CPE must contain part, vendor, product, and version information to identify a unique set of products. We can estimate the date when a vulnerability was created using the release date of the earliest affected product version.

NVD data is available for download in Extended Markup Language (XML) and JavaScript Object Notation (JSON) format; we used JSON files, which we read and parsed

in Python using the JSON module. We then extracted the data we needed such as CVE IDs, software and version data, and the date of publish.

2. Vendor Security Bulletins

We collected t_{patch} dates from vendor security bulletins or databases. Because CVE IDs are frequently used in cybersecurity, published security updates usually mention the CVE IDs that are corrected in the update. The vendors for the products in our dataset maintained central repositories for their security bulletins.

We extracted this information with Python's Scrapy module. Scrapy (Zyte, 2022) can design Web crawlers which navigate Web pages, follow links, and extract data from the Hypertext Markup Language (HTML) of each Web page. We also used the Selenium module (Selenium Project, 2022) to process dynamic Web pages that use JavaScript.

3. Exploit Database

To collect $t_{exploit}$ dates, we used the Exploit Database, a public database of exploits created for penetration testing (Offensive Security, 2022). The database contains approximately 50,000 exploits, some of which are tagged by CVE IDs to identify the vulnerabilities they exploit. Because we had trouble using its data extraction capability Searchsploit, we again used Scrapy to crawl the database and extract CVE IDs and the dates exploits were published.

4. Obtaining Version Release Information

After creating a list of affected versions for each CVE ID, we exported the complete set of versions for each software product line to a CSV (comma-separated value) file. We then searched the Internet manually to find the version-release dates. While many release dates were available through vendor websites, some minor version-release dates were obtained through third-party sources such as blogs.

IV. METHODOLOGY

A. PROCESS OVERVIEW

Figure 2 outlines the steps required to retrieve the data needed to determine $t_{creation}$, $t_{disclosure}$, t_{patch}, and $t_{exploit}$, which let us measure the perishability and obsolescence in software vulnerabilities. After downloading the JSON files for the years 2018–2021 from the NVD, we parsed them to retrieve the vulnerability IDs and their attributes (here called the CVE index). We also extracted the NVD publishing dates ($t_{disclosure}$), CVSS scores, and the affected operating system products and versions to aid our analysis. We used the product and version data to search online for vendor release dates, which were later entered in our dictionary and would be used to determine $t_{creation}$.

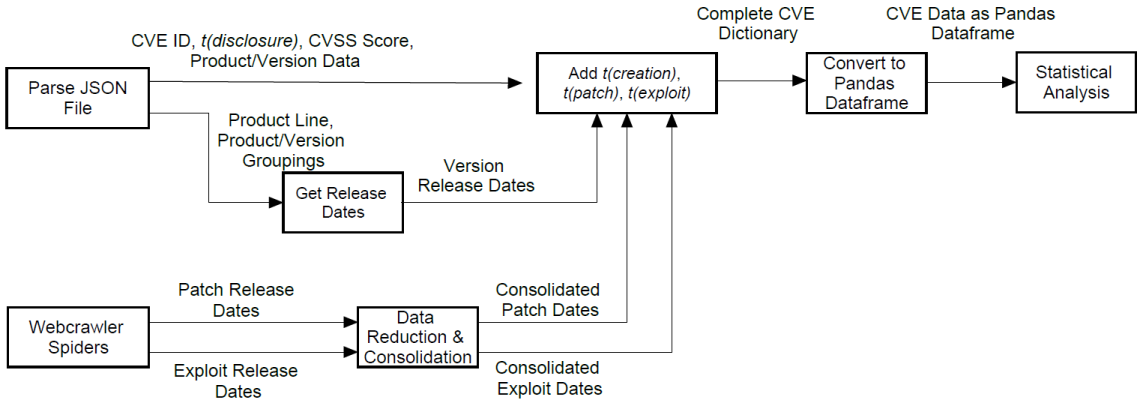

Figure 2. Data collection and analysis flowchart

As shown in Figure 2, we also programmed Web crawlers to extract the patch and exploit-publishing dates from vendor websites and the Exploit Database. The data reduction process consolidated individual vendor data into one file, and resolved multiple entries with the same CVE IDs by using the earliest date given. We then inserted our creation, patch, and exploit data into the CVE index, and exported the completed dictionary to a CSV file. The completed dictionary was then converted to a Pandas data frame for statistical analysis. Pandas (Pandas Development Team, 2020) is a Python package that allows data to be imported or converted into indexed data frames, permitting data

manipulation, slicing, and filtering; it can also handle missing data and data in date-time format.

B. PARSING JSON FILES

The JSON-format file from the NVD lists vulnerabilities and their attributes, including their CVE ID, CVSS score, and product identifications (CPEs). Figure 3 shows the structure and attributes of one such file, as displayed by Mozilla Firefox. We developed a Python script *jsonParse.py* that uses Python's JSON module to load the files, parse them, extract the necessary data elements, generate the vulnerability index, and export the data to CSV files. The process flowchart is shown in Figure 4. See Appendix A for the Python code.

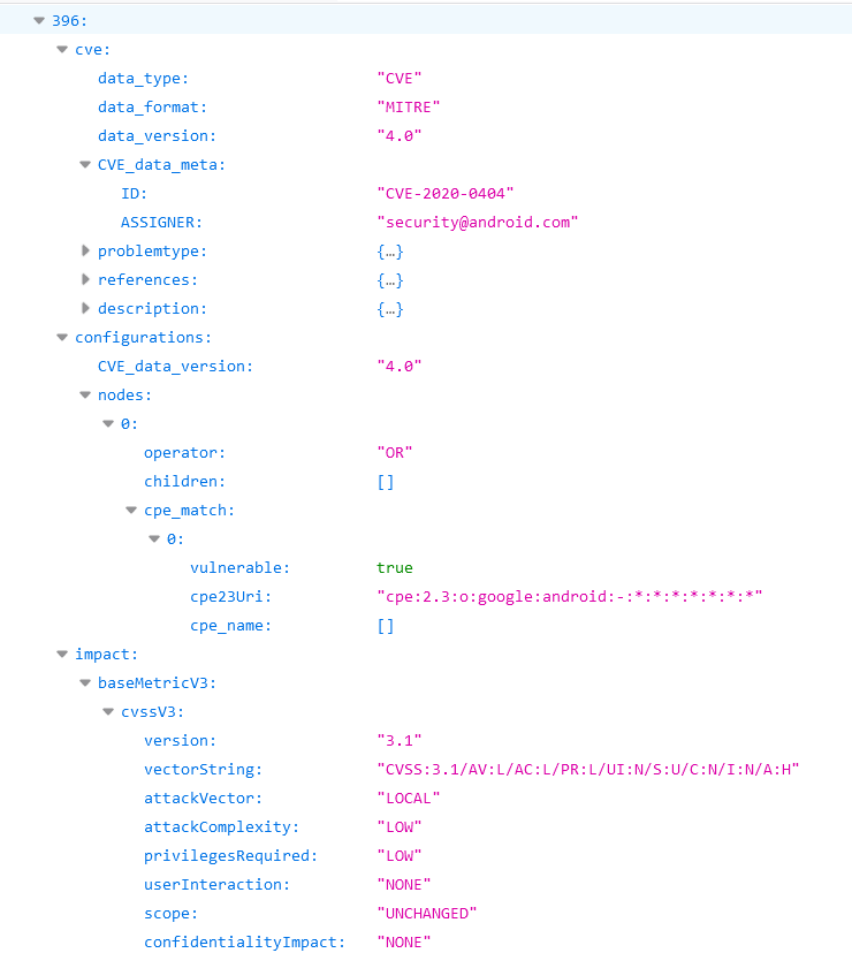

Figure 3. NVD JSON data, as viewed in Mozilla Firefox

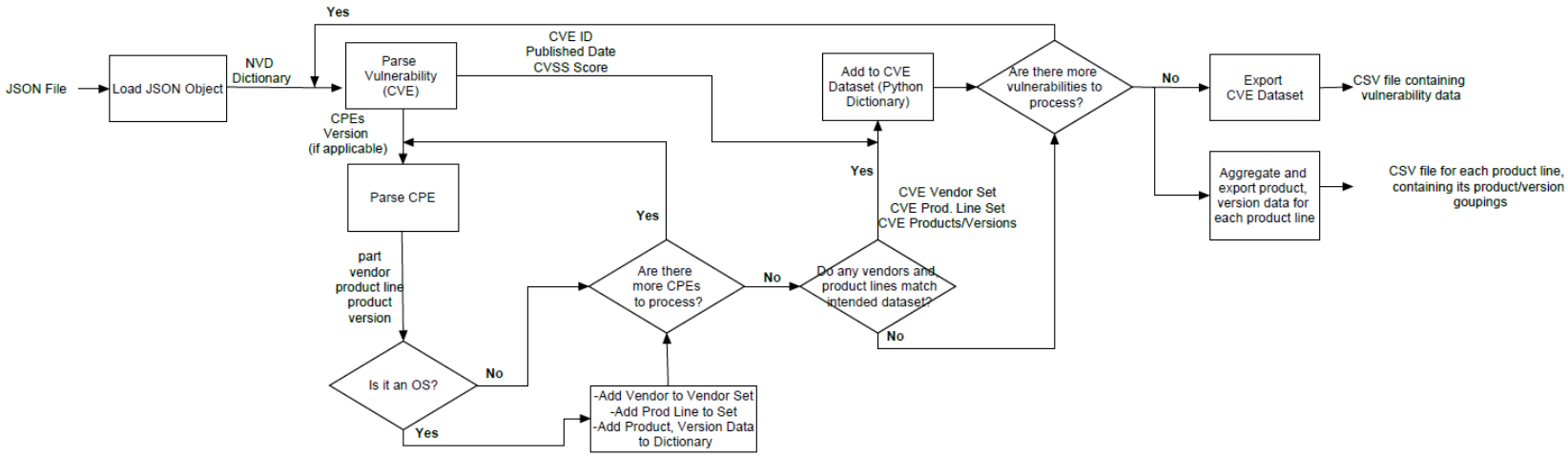

Figure 4. jsonParse.py data flow

Initially, our Python script used the JSON module to load each file and convert the contents to Python data structures that can be queried. Figure 5 shows the JSON elements we extracted. JSON objects are imported as dictionaries into Python, but the depth of the NVD files resulted in several layers of nested dictionaries, and we had to work down each layer to retrieve the desired data elements. CVE IDs, CVSS scores, and publishing dates were relatively simple to handle. Extracting vendor, product, and version information required further processing and parsing of the CPE field.

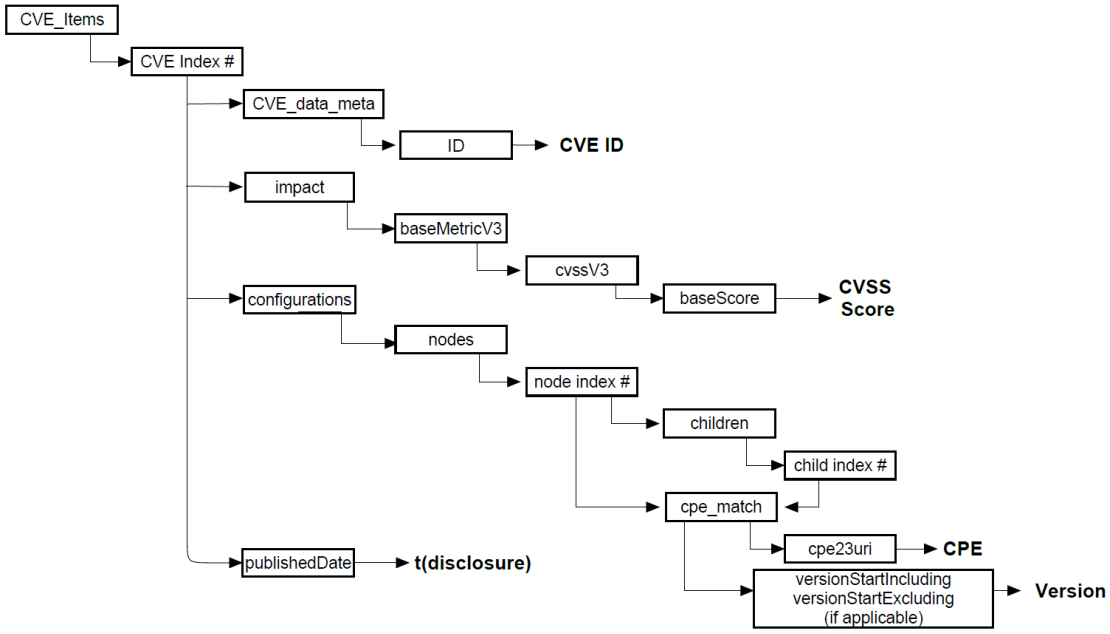

Figure 5. JSON file layers and extracted data elements

1. Parsing CPEs

Parsing the CPE product data enabled two things: We needed to determine if the product belonged in our dataset, and we needed the product and version data to extract software-release dates. The NVD cites alternative CPE specifications 2.2 and 2.3 to identify affected products; we chose CPE 2.3 to extract our data. (NIST, 2011) specifies that each CPE name consist of attribute-value pairs that describe its underlying product: The attributes we needed were *part*, *vendor*, *product*, and *version*. The pairs are represented

in the NVD as a string, with attributes separated by colons (see Figure 6). The *part* attribute has three classes: “h” (hardware), “o” (operating system), or “a” (application).

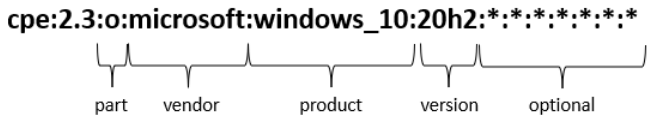

Figure 6. CPE name and attributes

Because each CPE product datum is a string, we could split it into its attributes at the colons. To determine if the product was in our dataset, we checked whether it belonged to the operating-system *part* class and if the vendor and product line were known vendors and product lines in our dataset. For product lines, we split the *product* value into words and assumed that the first word represented the product line (Figure 7).

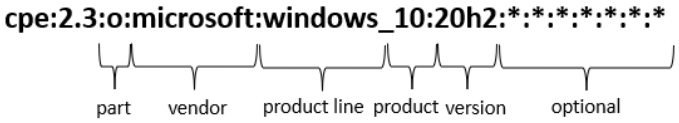

Figure 7. CPE name with *product* attribute split into product line and product

2. Retrieving Version Strings

Retrieving the strings for versions was more complicated because the NVD had two ways of indicating them. Sometimes the *version* attribute in the CPE contained a value, but for some vulnerabilities it was in a separate set of attributes in the JSON file (see Figure 5). The latter indicated a range of affected versions; we used the earliest affected version for which we had data.

3. Building the Vulnerability Index Data Structure

To build our initial vulnerability data structure, we used a Python nested dictionary; the key for each element in the dictionary was the CVE ID for that vulnerability. The value associated with it was another dictionary with the following key-value pairs:

- CVSS score: the CVSS score for that vulnerability
- $t_{(disclosure)}$: the date the CVE ID was published
- Vendor: vendors with products that were affected by the vulnerability
- Affected software: affected product lines
- Versions: key-value pairs where product line is the key, and value is a list of all affected software versions
- Keys for $t_{creation}$, t_{patch}, and $t_{exploit}$ were created and assigned "N/A" values to be populated later.

Using the csv Python module, we exported the product/version groupings for each product line to CSV files used to populate the version release dates. We also exported the CVE index to a CSV file for later processing.

C. POPULATING CREATION, PATCH, AND EXPLOIT DATES

For each product and version listed for a product line, we searched the Internet for the corresponding release date. We started by searching vendor websites; some, like Microsoft, kept a comprehensive set of release dates. Other vendor websites listed major version but not minor version releases such as a product being upgraded from 6.0 to 6.0.1. When the vendor website provided inadequate data, we searched blogs, bulletins, and other sources of information. If subsequent versions were affected, we used the release date for the first version dated after it.

1. Creation and Patch Dates

Windows patch release dates were obtained from the Microsoft Security Response Center website (https://msrc.microsoft.com/update-guide). The exported CSV file included the vulnerabilities patched by each update. To get patch data for the other product lines, we used the Scrapy module (Zyte, 2022) to crawl the websites where vendors published their security bulletins. Scrapy retrieves specified HTML documents with "spiders," queries their contents, follows links, and extracts desired data. The Python code is in Appendix B. Each website has a different structure, but generally the steps were (Figure 8):

1. Send HTML GET requests for the starting pages.
2. Query the retrieved HTML file for security-bulletin links and the dates they were published.
3. Iterate through the links and their publishing dates. If the date for a bulletin was within the scope of our dataset, the bulletin was retrieved. The spider ignored pages published before 2018.
4. Query the HTML file for the bulletin and extract the CVE ID and publishing date.
5. Export the vulnerability ID and publishing date to a CSV file.
6. Navigate to the next page.

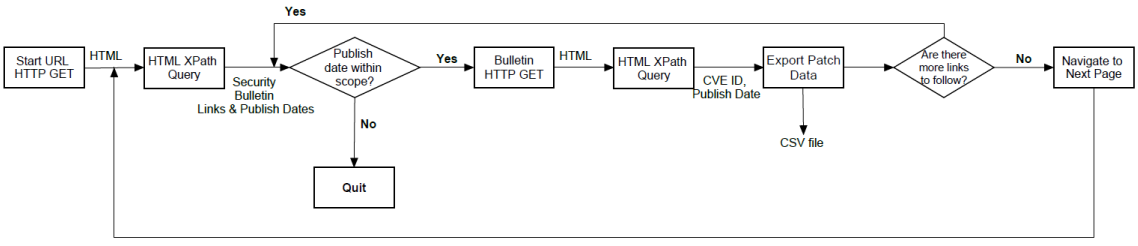

Figure 8. Spider process for extracting CVE IDs and patch publish dates. This could vary based on the structure of the website.

Programming the spiders to query the HTML contents required first checking the target HTML pages; we used Mozilla Firefox's Inspect function to determine which page elements contained the data we needed. Our first spider used the BeautifulSoup module (Richardson, 2015) to crawl the Google Chrome release blog (https://chromereleases.googleblog.com/), but our subsequent crawlers used Scrapy's XPath selectors. While BeautifulSoup can query the contents of Google's blog posts about patch releases, we found the XPath syntax better for querying the other pages. Many used multiple HTML elements such as tables without unique attributes, and only some contained data relevant to our analysis. XPath enabled more precise queries for our spiders.

CVE IDs are widely used in security bulletins and software-patch updates to identify vulnerabilities that were corrected. For each bulletin, we extracted the date it was published (the patch release date) and any CVE IDs and wrote this to a CSV file for each vendor. For Fedora, which has an open-source development and update system (https://bodhi.fedoraproject.org/), we only used the date on which a build was declared stable as the patch date. Some websites were harder to crawl than others due to their structure. For example, Redhat used JavaScript on its customer portal which Scrapy cannot process by itself (3i Data Scraping, 2021). To handle this issue, we used the Selenium Python module (Selenium Project, 2022). Selenium is commonly used for web testing; it uses a "webdriver" browser engine (either Gecko or Chrome) to load pages which can handle JavaScript (3i Data Scraping, 2021). Using it with Scrapy, we could retrieve the page source code and query it as we did for the other spiders. For other websites, Scrapy erroneously treated some links as duplicates and filtered them out (Scrapy Developers, 2022); we changed the setting within the spider to disable filtering.

2. Exploit Dates

To get dates that exploits were published, we first tried to use Searchsploit on an Ubuntu virtual machine; it retrieves data from the Exploit Database and can output to a JSON format (Offensive Security, 2022). However, the output listed the publishing date for all exploits as zero in epoch time (01/01/1970), which was obviously an error. So instead, we extracted the exploit publishing dates and associated vulnerabilities using the

same process as for patch data, by using another spider to crawl the Exploit Database website.

Our spider encountered several problems. Initially, HTTP GET messages returned a code 403 (Forbidden) status code, which we corrected by adding Mozilla Firefox header data to the settings for our spider (Mamka, 2016). Once we got it crawling, the Web server sometimes spontaneously terminated the TCP (transfer control protocol) connection. We suspect that this was a response to the large volume of requests we were sending, which may have looked like a denial of service (DoS) attack. We fixed this problem by adjusting the spider settings to delay successive requests by one second, with the drawback that it took longer to crawl the database.

When following links on the page, the Web server would sometimes redirect us back to the main page instead. This was possibly a JavaScript issue. We corrected this problem by using full page links (URLs). The page URLs on this site were identical except for the exploit serial numbers which were generally sequential integers, although some numbers were skipped. So we just tried to retrieve pages in numerical sequence, which worked because the spider would skip over pages generating an HTTP error code 404 (Not Found) response and move on to the next link.

3. Data Reduction and Consolidation

Following data extraction, we observed multiple entries with the same CVE ID, most likely due to multiple patches correcting the same vulnerability for different products. We removed the later such instances. We then consolidated the patch data into a single file using a Python script we wrote, *dataReduction.py*. The consolidated CVE index was then written to a master output file, yielding one CSV master file for patch dates and one for exploit dates.

4. Aggregating Creation, Patch, and Exploit Data into the CVE Index

The last step copied the creation, patch, and exploit dates to the CVE index, giving us $t_{creation}$, t_{patch}, and $t_{exploit}$ using another Python script we wrote (*integrateCVE.py*). For patches and exploits, the script would read each row of our CSV master files into a list,

then iterate through that list and check if the vulnerability was in the CVE index. If it was, the associated date was assigned as either t_{patch} or $t_{exploit}$.

Because the creation dates were assigned by product/version groupings and not by CVE ID, our Python script had to check if each product line and product/version grouping was listed for each vulnerability in the CVE index. If it was, and a $t_{creation}$ value was not yet assigned, the creation date for that software version was assigned as $t_{creation}$ for that vulnerability. If a $t_{creation}$ value was already assigned, for those vulnerabilities in more than one product line or in more than one version within a product line, the earlier date was used. We also extracted CVSS scores to analyze differences between vulnerabilities based on their severity. Finally, we exported the now-completed CVE index to a new CSV file.

D. STATISTICAL ANALYSIS

Once we accumulated and aggregated our data, we converted the Python dictionary to a data frame using the Pandas package. Pandas (Pandas Development Team, 2020), also called the Python Data Analysis Library, is a data analysis tool that provides ways to view and manipulate data including dates. This was ideal for our analysis, which needed to calculate the duration (the difference) between dates (such as t_{patch}-$t_{disclosure}$ to find the duration from disclosure to patch release). To plot our results, we used the Matplotlib Python package. Matplotlib (Caswell et al., 2021) can display data in scatter plots, histograms, and cumulative distribution functions.

V. ANALYSIS OF RESULTS

A. DATA ANALYSIS: ALL VULNERABILITIES

We successfully collected 10,912 operating system vulnerabilities from the NVD. Because all vulnerabilities had publishing dates, we extracted $t_{disclosure}$ for the entire set of vulnerabilities. Of these, we found $t_{creation}$ for 7,893 vulnerabilities, t_{patch} for 8,860 vulnerabilities, and $t_{exploit}$ for 322 vulnerabilities. Once the data was aggregated into one CSV file, we imported it into a Pandas data frame with Python, which permitted us to calculate the duration between the dates to determine the lengths of the phases of their life cycles. The four durations we were interested in were the time to disclosure (Δ_{cd}), the time to patch (Δ_{dp}), the time to exploit (Δ_{de}), and the longevity (Δ_{cp}). Δ_{cd} represents the duration from $t_{creation}$ until its $t_{disclosure}$, while Δ_{dp} and Δ_{de} are the durations from disclosure until patches and exploits, respectively, were published. Δ_{cp} represents the total lifespan of a vulnerability, from $t_{creation}$ to t_{patch}. Table 1 contains the statistics for each period for the dataset.

Table 1. Vulnerability statistics - overall

	Vulnerabilities Observed	Mean (days)	Median (days)	Standard Deviation	25th Percentile	75th Percentile
Time to Disclose (Δ_{cd}) ($t_{disclsoure}$ - $t_{creation}$)	7893	1697.72	1364	1334.21	708	2371
Time to Patch (Δ_{dp}) (t_{patch} - $t_{disclosure}$)	8860	-11.35	-1	111.4	-14	8
Time to Exploit (Δ_{de}) ($t_{exploit}$ - $t_{disclosure}$)	322	-9.57	1	113.84	-23.75	7
Longevity (Δ_{cp}) (t_{patch} - $t_{creation}$)	7275	1737.65	1410	1366.51	665	2600

1. Longevity

We first observed Δ_{cp} using the subset of vulnerabilities for which we had both $t_{creation}$ and t_{patch}. We found that the median lifespan of a vulnerability was 1,410 days, or approximately 3.86 years. We also found a high degree of dispersion among the values:

The bottom quartile was patched in less than two years (665 days), while the top quartile was patched after more than 7.1 years (2,600 days), which can be seen in the histogram (Figure 9). The CDF (Figure 10) shows that when an operating system was first released, over 70% of vulnerabilities were patched within the first five years (1,825 days). After that, the rate of patching slowed considerably.

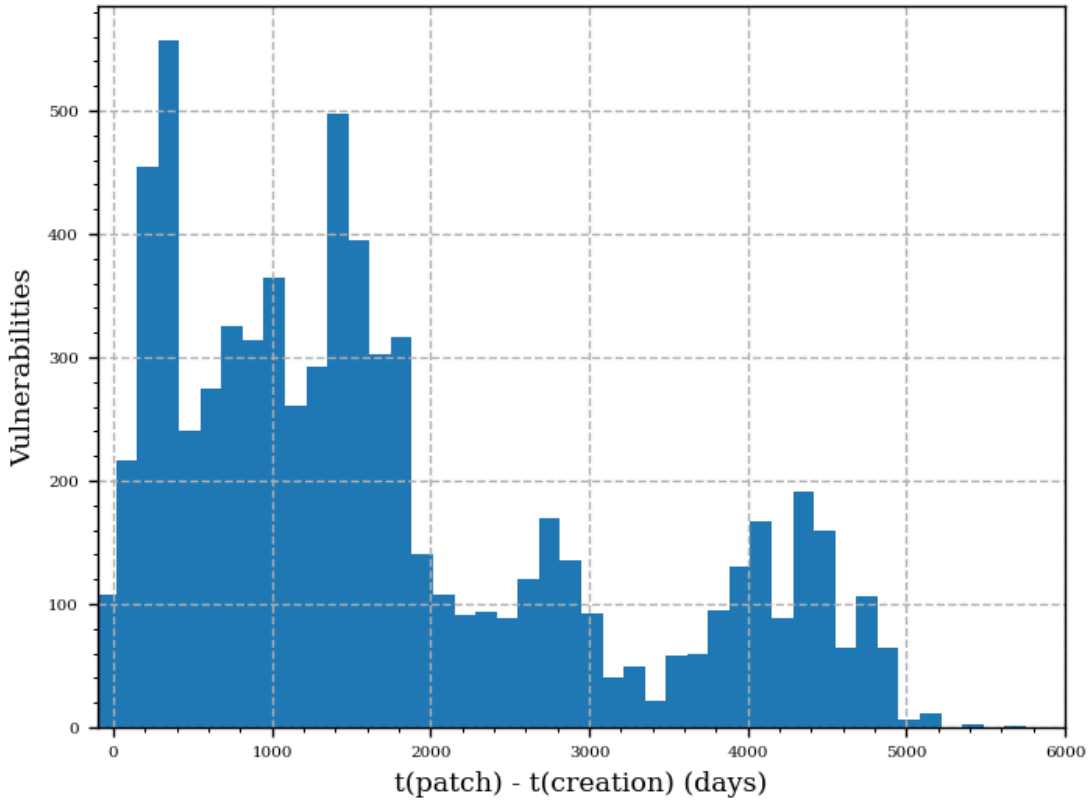

Figure 9. Vulnerability longevity (Δ_{cp}) histogram

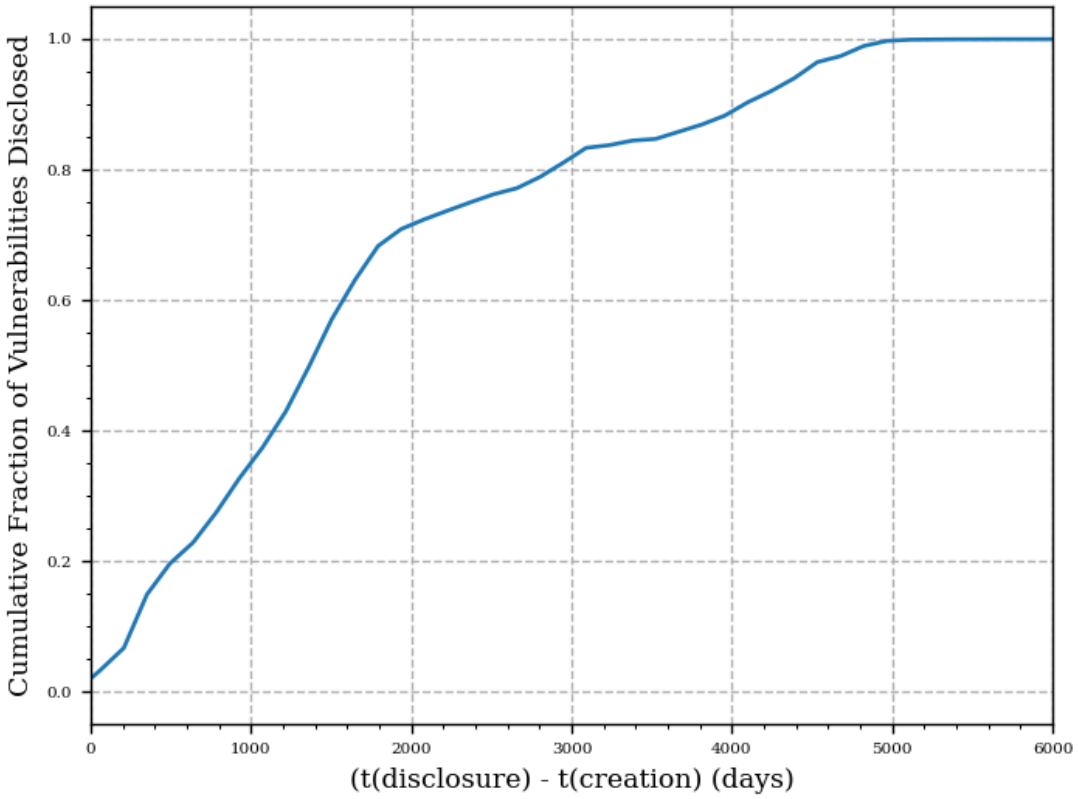

Figure 10. Vulnerability longevity (Δ_{cp}) CDF

Interestingly, 76 vulnerabilities had a longevity that was negative in that the patch was released before the product containing the vulnerability. Further examination of those vulnerabilities indicated that, in some cases, the patch was released by a developer to fix third-party software before the release of the affected operating-system version; the operating system was probably released with old, unpatched software installed or bundled with it. In other cases, multiple operating systems were affected by the vulnerability, but the date of earliest affected version was unavailable. Other cases could not be easily explained from the data in the NVD and in security bulletins from the operating system vendor; several of these cases involved the Fedora operating system.

2. Kaplan-Meier Survival Analysis

We used survival analysis to better understand the probability of a vulnerability reaching a specified longevity. Survival analysis is a form of statistical analysis that

measures the duration of time leading up to the occurrence of an event (Kleinbaum & Klein, 2005). This method is often used for clinical studies in medicine but can be used for many other purposes, and it can handle censored data (where some but not all the data for all subjects has been observed), permitting us to include in our dataset those vulnerabilities that have not yet been patched (right-censored). The survival function $S(t) = P(T > t)$ indicates the probability that an event (with survival time T) will occur after a specified time t. We used the Python Lifelines package (Davidson-Pilon, 2019) to plot the survival function for the dataset. The Lifelines package takes a Pandas data frame as input and fits the data to the survival function.

We plotted the Kaplan-Meier curve (Kleinbaum & Klein, 2005) first because it is non-parametric (makes no assumptions about the data) (Ismiguzel, 2021). This curve is generated using a function known as the Kaplan-Meier estimator, which estimates the survival probability over a specified period (Lewinson, 2020). The formula for this estimator is $\hat{S}(t) = \prod_{i:t_i \leq t}(1 - \frac{d_i}{n_i})$, where d_i represents the number of events that have occurred and n_i the number of surviving elements at time t_i. Kleinbaum and Klein describe the probability of survival past time t as the product of the survival probability at each previous time an event occurs and the probability of surviving at time t, given survival up to time t. Applying this definition to our longevity data, the probability of a vulnerability surviving past time t using the Kaplan-Meier estimate is the probability of the vulnerability being patched at time t (given that it survived until time t), multiplied by the product of the survival probabilities of all vulnerabilities that were patched before it. The Kaplan-Meier curve for our data is in Figure 11.

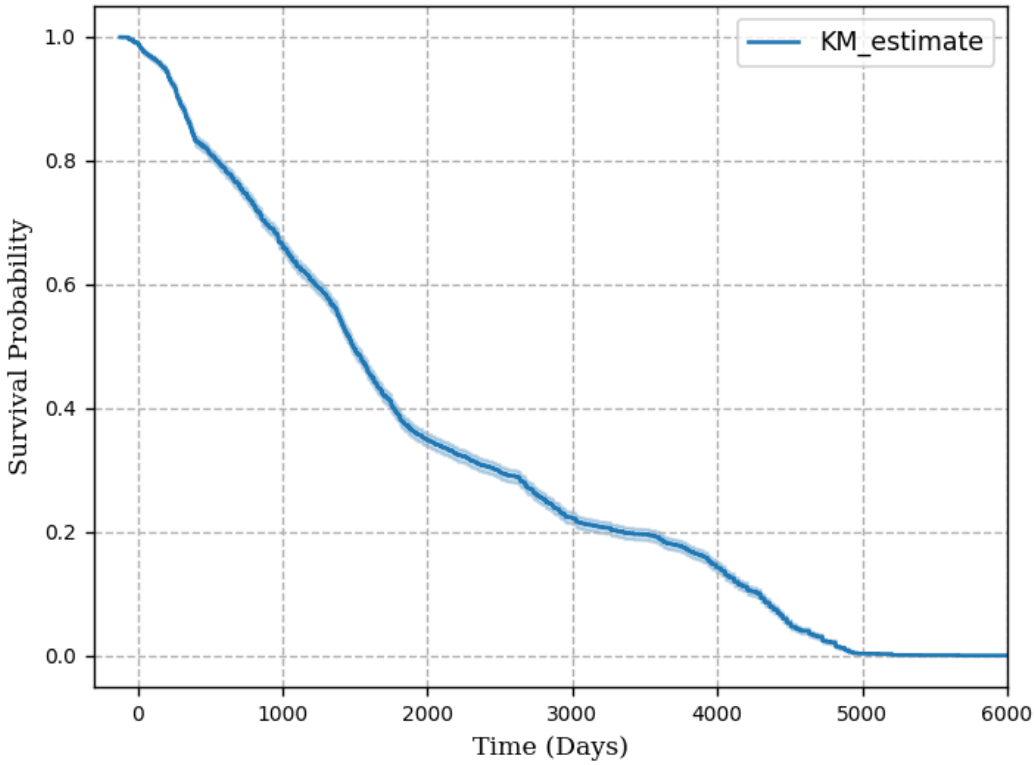

Figure 11. Kaplan-Meier survival curve for vulnerability longevity (Δ_{cp}), with 95 percent confidence intervals

The median survival time was 1,485 days, with a 95 percent confidence interval of 1467 to 1517. That is higher than the 1,410 days observed using the uncensored data, suggesting that several vulnerabilities have remained unpatched for years, possibly because the software is no longer supported. The survival function shows a steep drop in the survival rate out to approximately 2,000 days; the probability a vulnerability will remain unpatched past this point is less than .35. Then the survival rate declines more slowly; at ten years (3,650 days), the probability of survival is still .18. This is consistent with what we observed in the CDF in Figure 10.

3. Survival Analysis Using Parametric Functions

We also fit a parametric distribution model to our longevity data. We tried both a Weibull distribution (Reliasoft, 2002) and an exponential distribution (Zach, 2021) based on the shape of the CDF. The Lifelines package supported these parametric distributions,

and like the Kaplan-Meier curve, data with missing parameters could be included, although we could only include vulnerabilities with positive longevity values. After plotting both CDFs against the CDF for the raw data, we determined that the Weibull distribution fit the best (Figure 12). A Weibull distribution has survival function $S(t) = e^{(-(\frac{t}{\eta})^{\beta})}$ and a probability density function $f(T) = \frac{\beta}{\eta}(\frac{T-\gamma}{\eta})^{(\beta-1)}e^{(-(\frac{T-\gamma}{\eta})^{\beta})}$, where β represents the shape, η represents the scale, and γ represents the location (Reliasoft, 2002). Figure 13 shows the survival function for the Weibull distribution we fit to our longevity data, with η = 2033.55, β = 1.26, and γ = 0.

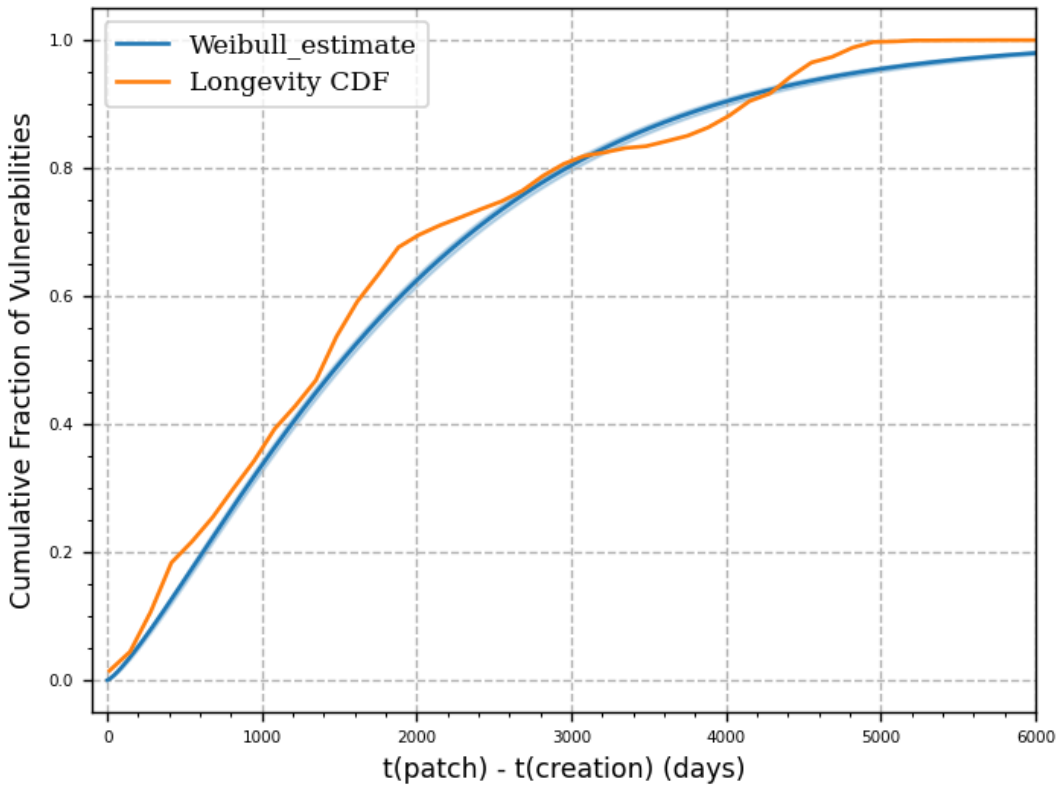

Figure 12. Weibull CDF laid over longevity CDF

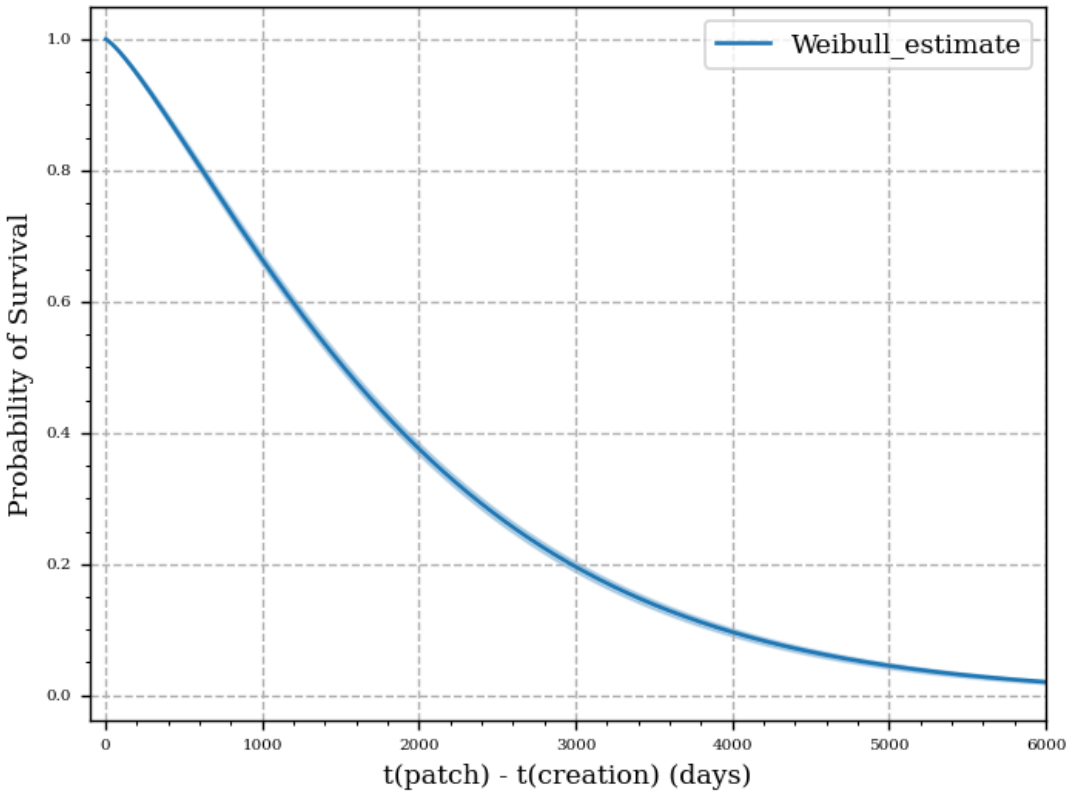

Figure 13. Weibull survival function for vulnerability longevity

B. ASSESSMENT OF VULNERABILITY LIFE CYCLE PHASES

Besides the overall lifespan of a vulnerability, we assessed the durations of three phases: time to disclose (Δ_{cd}, $t_{disclsoure}$ - $t_{creation}$), time to patch (Δ_{dp}, t_{patch} - $t_{disclosure}$), and time to exploit (Δ_{de}, $t_{exploit}$ - $t_{disclosure}$).

1. Vulnerability Time to Disclosure

We observed a median time to disclosure of 1,364 days (3.74 years) and a mean of 1,697.72 days (4.65 years). As with the longevity results, the standard deviation was high (1,334 days), indicating considerable variability within the dataset. The bottom quartile of vulnerabilities would be disclosed in less than two years (708 days), while the top quartile would require more than 6.5 years (2,371 days). As shown in Figures 14 and 15, the histogram and CDF for the time to disclosure look like those for the longevity dataset,

indicating that a vulnerability will probably spend most of its life in an undisclosed (zero-day) state.

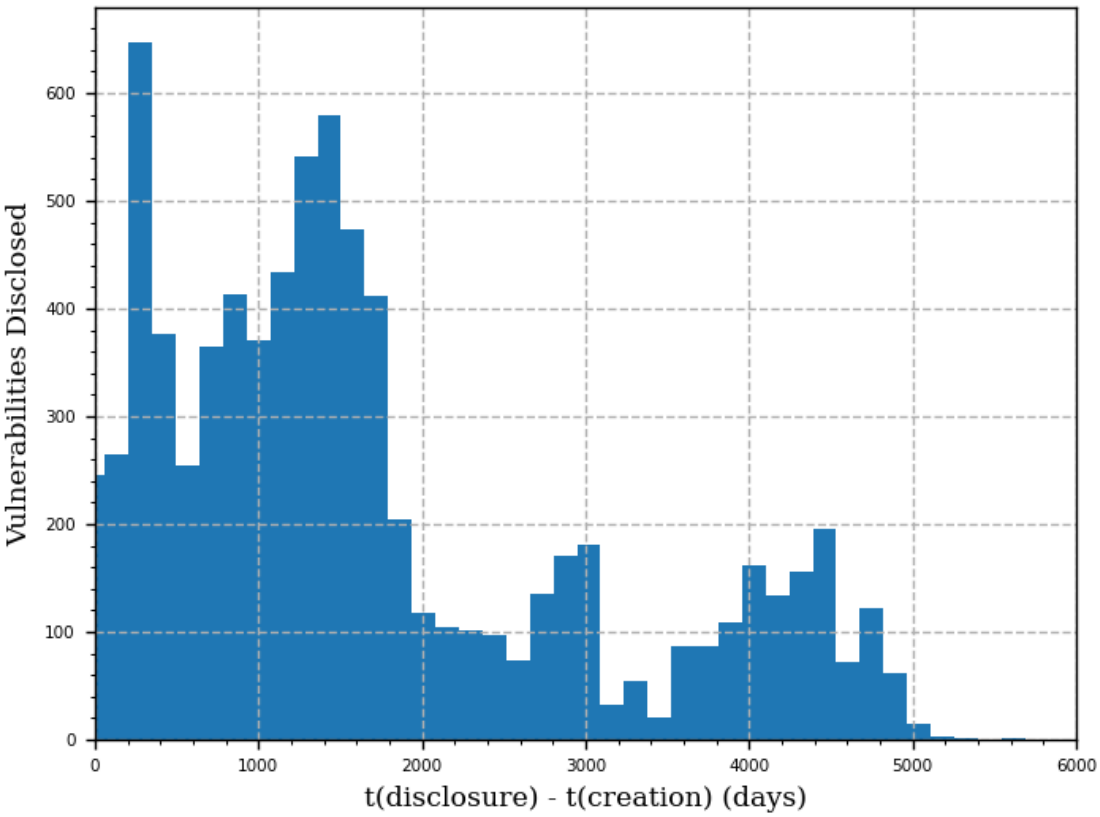

Figure 14. Histogram for time to disclose (Δ_{cd})

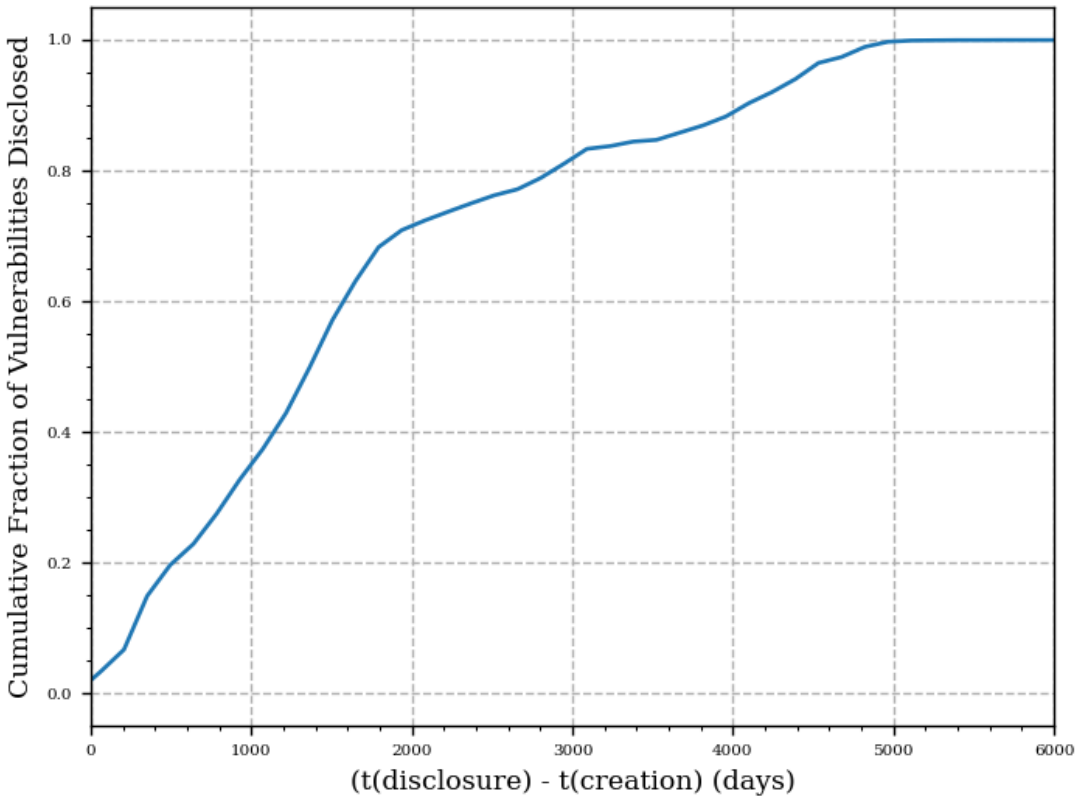

Figure 15. CDF for time to disclose (Δ_{cd})

2. Vulnerability Time to Patch

We next observed Δ_{dp} using the subset of vulnerabilities with t_{patch} values and compared them to the date of disclosure. The median was -1 days and only 35.68 percent of patches were released after disclosure. This suggests that white-hat hackers and security researchers often report vulnerabilities to vendors before disclosing them ("responsible disclosure") (Sen et al., 2020). Some vendors provide bug bounties or other incentives to encourage this behavior, which gives them time to develop a patch before the vulnerability is disclosed (Microsoft, 2022).

Most patches were released within weeks of the date of disclosure; the range of the middle quartiles (25th -75th percentile) was [-14, 8] days. This concentration near the disclosure time is clear in the histogram (Figure 16) and the CDF (Figure 17). However,

the data becomes more widely dispersed as the time before or after disclosure increases, resulting in a standard deviation of 111.4 days.

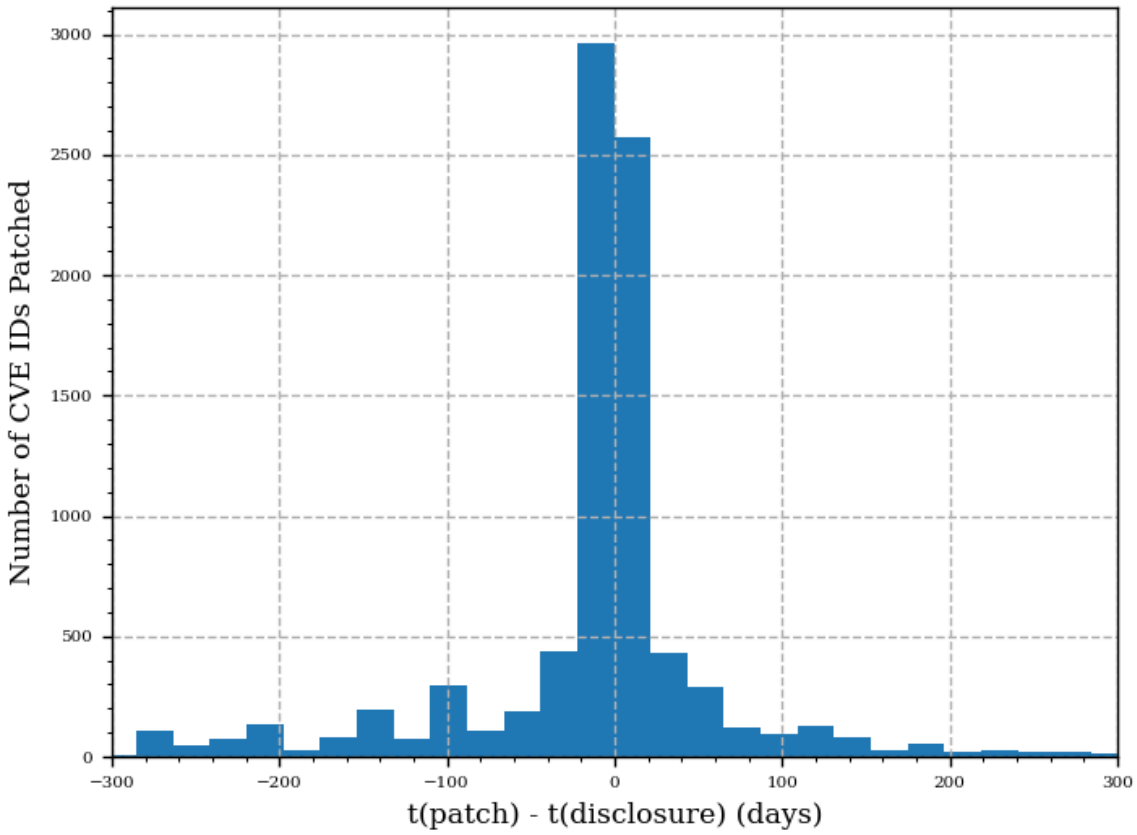

Figure 16. Histogram for time to patch (Δ_{dp})

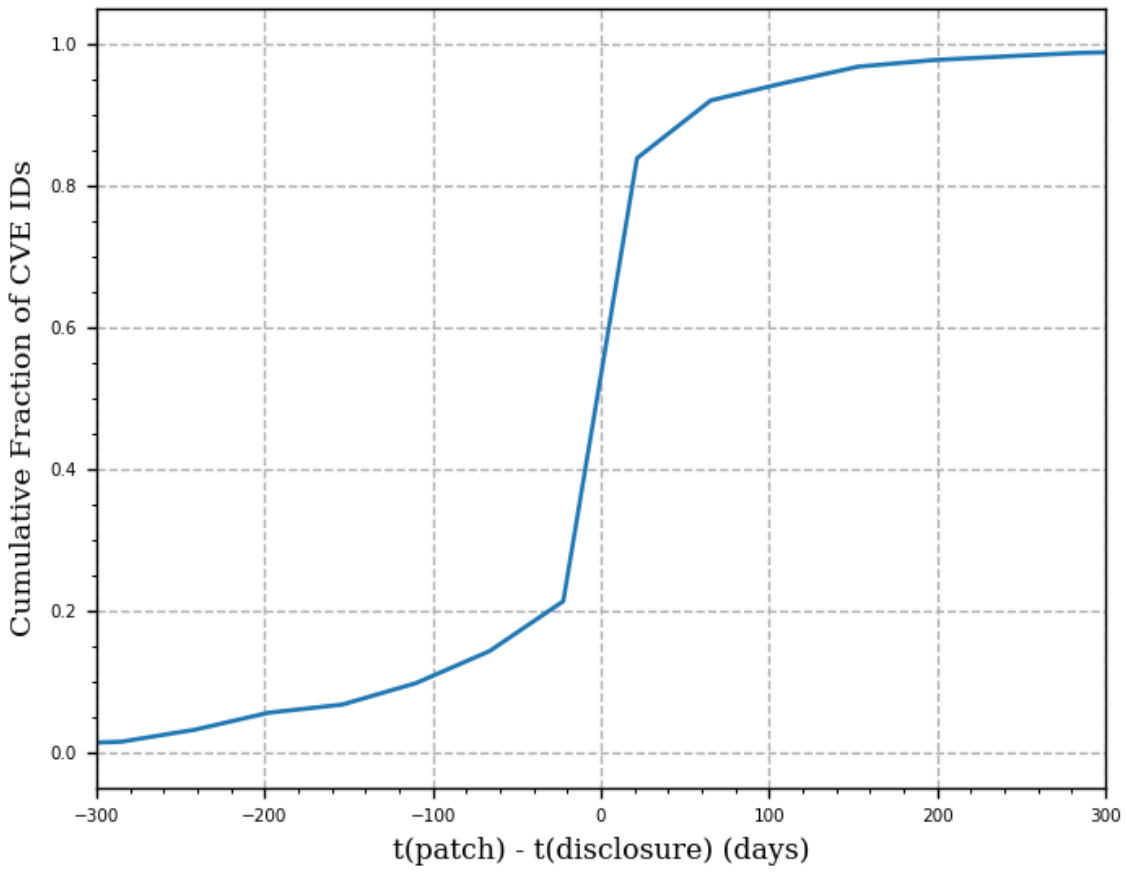

Figure 17. CDF for time to patch (Δ_{dp})

3. Disclosure to Exploit Availability

The data for vulnerabilities with known exploit dates was much smaller than the others; only 322 vulnerabilities in our dataset had them. While most patches were released on or before disclosure, the opposite was true for exploits; 58 percent of exploits were released after their vulnerability had been disclosed. As with time to patch, the histogram for Δ_{de} (Figure18) shows that most exploits are published close to disclosure. The CDF for Δ_{de} (Figure 19) shows a steady rise in the share of exploits published until just before disclosure. Then a strong upward trend starts and more exploits are published.

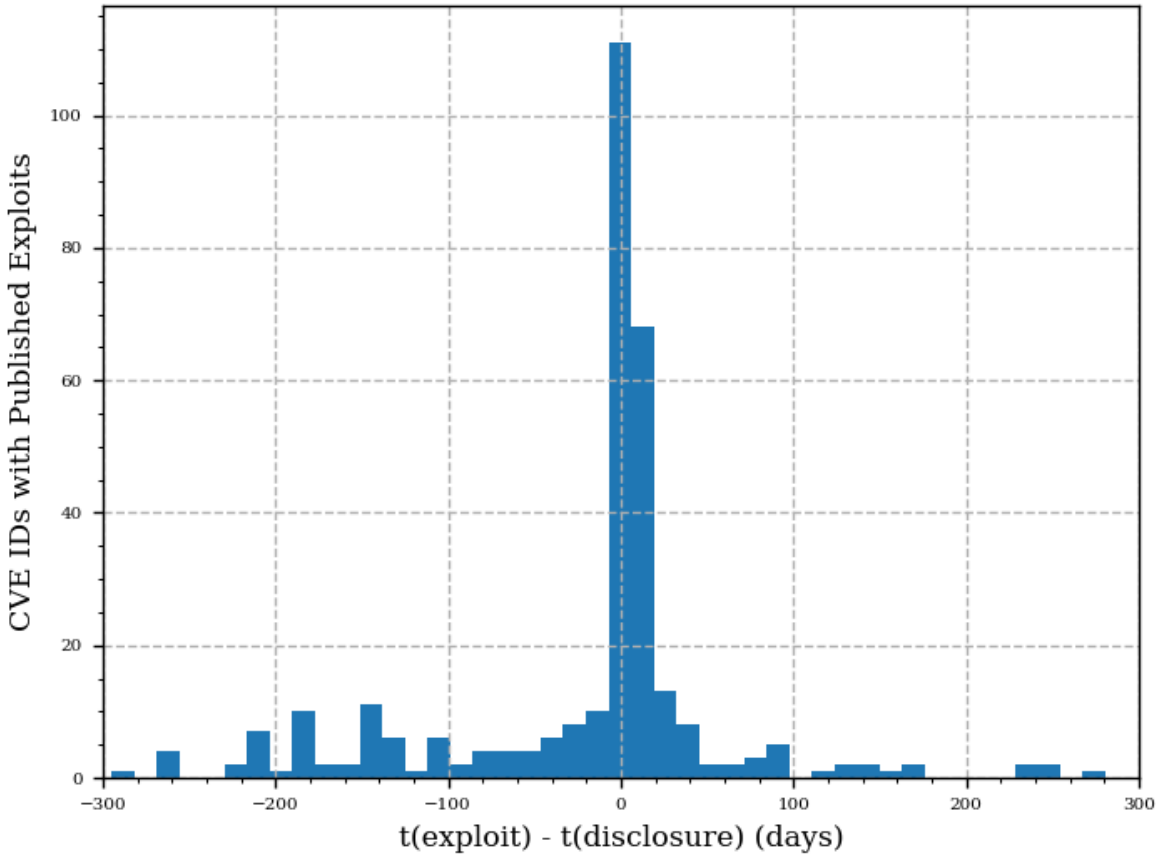

Figure 18. Histogram for time to exploit (Δ_{de})

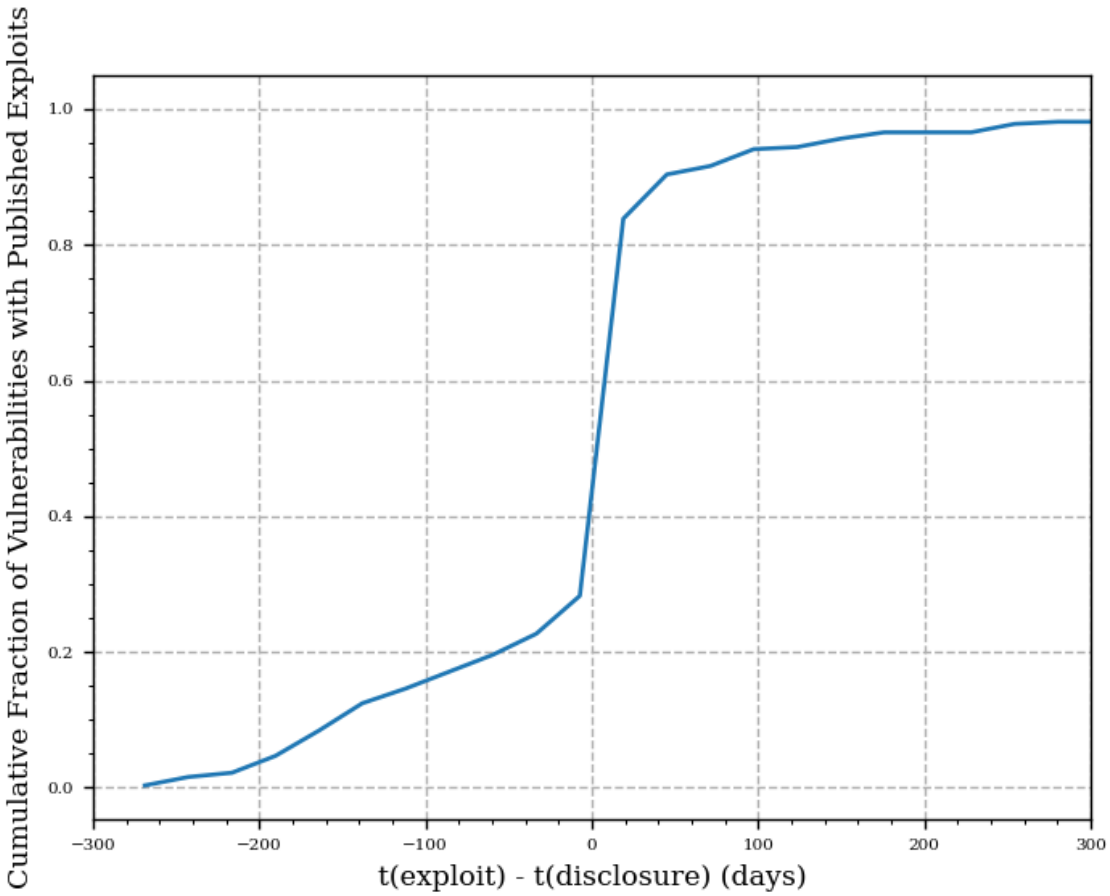

Figure 19. CDF for time to exploit (Δ_{de})

Two reasons may explain why most exploits are released after disclosure. First, the type of actors that populate the Exploit Database include many who do vulnerability research and penetration testing (Offensive Security, 2022), white-hat actors willing to cooperate with vendors to privately disclose vulnerabilities. Second, many exploits are probably developed after disclosure of vulnerabilities with reverse engineering of patches (Frei et al., 2006).

C. FACTORS THAT COULD AFFECT VULNERABILITY LIFESPAN

We also examined some factors to determine if they affected the phases of the software vulnerability life cycle, resulting in a lifespan that differed from those of the overall dataset. These are also considerations for cyber-operations planning.

1. Known Existence of an Exploit

We tried measuring the effect of having a known exploit on vendor patching, using data in the Exploit Database. We assumed that the reaction from vendors to exploits published by white-hat actors and penetration testers would suggest their response to malicious exploits. We would then determine how the discovery of an exploit by a victim impacted the time to patch, Δ_{dp}, or the longevity, Δ_{cp}. We considered only those vulnerabilities with exploits released before disclosure ($\Delta_{de} < 0$). Of 116 vulnerabilities that met this criterion, 110 were patched, while six remained unpatched. However, we discovered that most vulnerabilities were still being patched before they were exploited; the median for Δ_{de} was -81 days while the median for Δ_{dp} was -101.5 days. It became clear that the data did not show what we were trying to measure. We tried considering only vulnerabilities where the exploit was released before the patch ($\Delta_{de} < \Delta_{dp}$), but only 12 vulnerabilities fit this criterion. All but one was disclosed within 20 days after the exploits were published; the delay in disclosure for the remaining exploit was 156 days. Therefore, we could not draw any conclusions about this issue.

2. Observing Patching Behavior Where Coordination Is Unlikely

We tried to determine how long it took vendors to react when a vulnerability was disclosed of which they were unaware. To measure this, we used the subset of vulnerabilities where $\Delta_{dp} > 0$. We assumed that for this subset of vulnerabilities, coordination between white hats and vendors was unlikely because a vendor would not want a vulnerability to be disclosed until they could develop and release a patch.

We narrowed our dataset to 3,161 vulnerabilities where $\Delta_{dp} > 0$. The median time to patch was 18 days, while the mean was 57 days, so some vulnerabilities took a long time to patch after they were disclosed. As with longevity, we tried to fit a survival function to this dataset. Based on the shape of our model, the most likely distributions that might fit were exponential, pareto, and Weibull. Weibull was the best fit for this curve with $\beta = .70$, $\eta = 42.48$ and $\gamma = 0$; the survival function is shown in Figure 20 and the CDF overlay is shown in Figure 21. The median survival time was 25.09 days; there is a 50 percent chance a vulnerability will survive past this point.

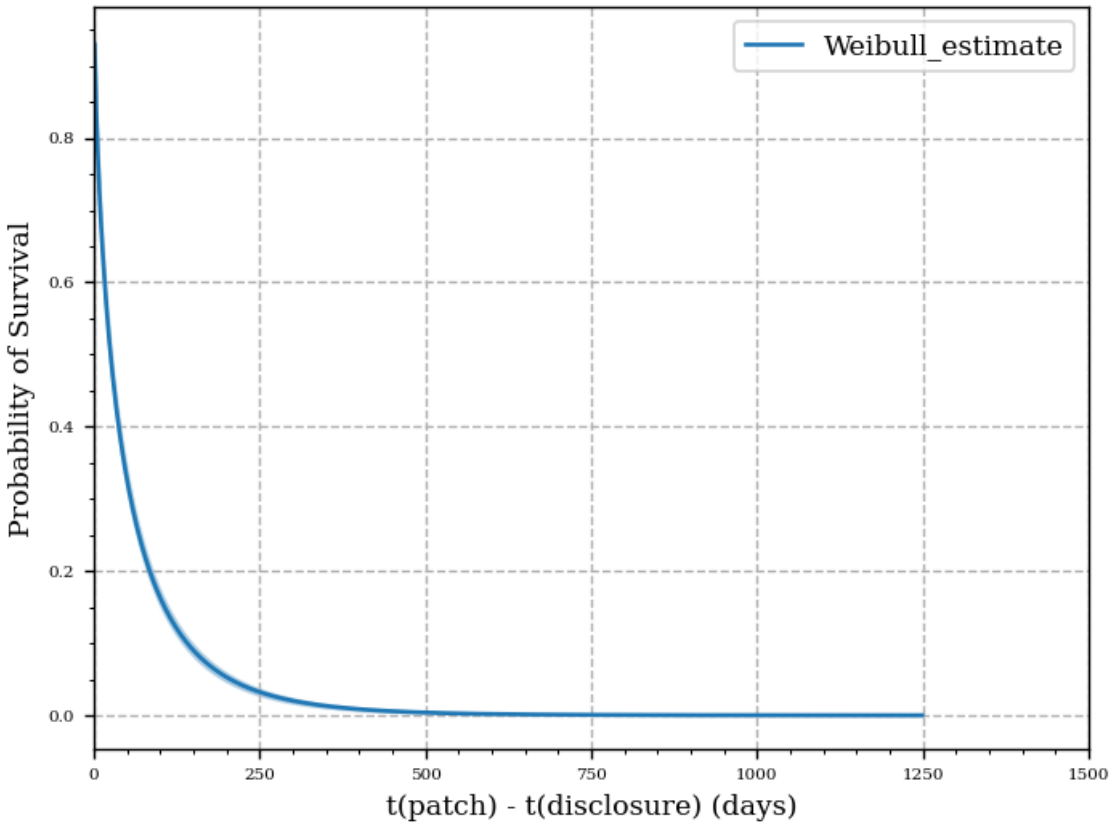

Figure 20. Weibull survival function for time to patch where $\Delta_{dp} > 0$ (uncensored data only)

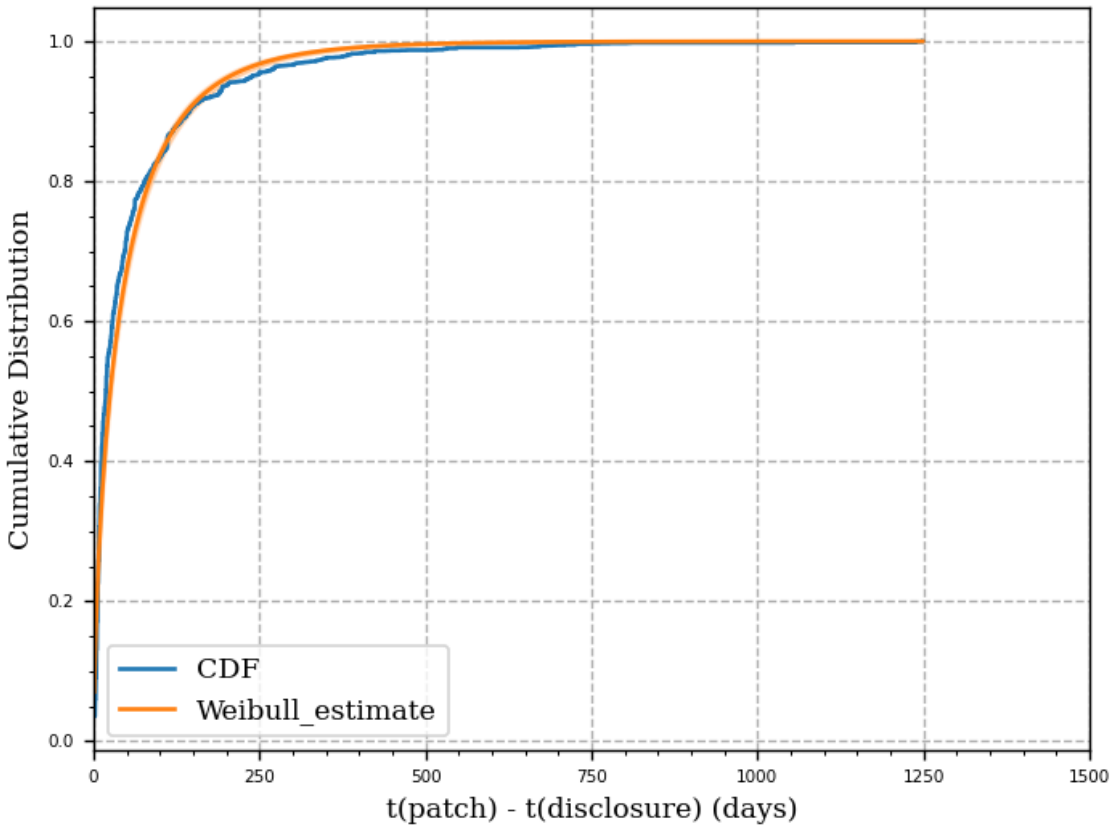

Figure 21. CDF for time to patch (Δ_{dp}) overlaid with Weibull model CDF

Unlike with longevity, this Weibull distribution model used only uncensored data because from a cyber-operations planner's perspective, a deployed cyberweapon will most likely be patched once discovered. Therefore, it is more useful to understand the time it took for vendors to release patches when they felt they needed to do so, rather than considering those that were not patched.

3. Vulnerability Severity

We also sought to determine any significant differences in the software vulnerability life cycle based on the severity of a vulnerability. CVSS scores are segregated into four levels of severity: Low (0-3.9), Medium (4.0-6.9), High (7.0-8.9), and Critical (9.0-10.0) (NIST, 2022). Table 2 contains the statistical results for each level. We noticed that our dataset contained very few Low-severity vulnerabilities; this could be because they pose so little threat that many vendors do not bother to disclose them for inclusion in the

NVD. Most vulnerabilities were either Medium- and High-severity, although the population of Critical vulnerabilities was sizable. For most phases of the software vulnerability life cycle, the medians for each level were similar, except for Δ_{de}, for which we had less data.

Table 2. Vulnerability statistics – by CVSS severity

Time to Disclose: $t_{disclsoure} - t_{creation}$

Severity	Number Observed	Mean	Std Dev	Median
Low	189	1592.93	1325.23	1270
Medium	2876	1597.35	1237.21	1359
High	4111	1800.33	1433.76	1375
Critical	717	1539.63	1041.54	1375

Time to Patch: $t_{patch} - t_{disclosure}$

Severity	Number Observed	Mean	Std Dev	Median
Low	233	-6.66	139.68	0
Medium	3237	-3.65	125.09	0
High	4714	-16.81	98.33	-1
Critical	676	-11.81	114.6	0

Time to Exploit: $t_{exploit} - t_{disclosure}$

Severity	Number Observed	Mean	Std Dev	Median
Low	4	16.5	17.54	10
Medium	64	33.7	151.79	4
High	220	-21.91	102.68	1
Critical	34	-14.21	85.47	1.5

Longevity: $t_{patch} - t_{creation}$

Severity	Number Observed	Mean	Std Dev	Median
Low	164	1711.25	1390.29	1459
Medium	2648	1636.21	1266.07	1405
High	3877	1828.43	1457.67	1407
Critical	586	1602.8	1108.97	1427

The CDFs for longevity are shown in Figure 22; for approximately the first 1,500 days, the CDFs track closely together, with over half being patched during that period, then they begin to diverge slightly. It seems that High-severity vulnerabilities were patched more slowly, and Critical-severity vulnerabilities were patched more quickly than the

others, although all four CDFs track closely to one another. When we collapse the severity levels from four to two (Low/Medium and High/Critical), as shown in Figure 23, we see the same pattern where vulnerabilities are patched at the same rate through approximately 1,500 days before diverging. After 1,500 days, more severe vulnerabilities have slightly longer lifespans (are patched more slowly) than less severe ones, not what we expected. We had expected that more severe vulnerabilities would be discovered and patched faster due to the greater risk that they pose. This suggests they are harder to find and patch.

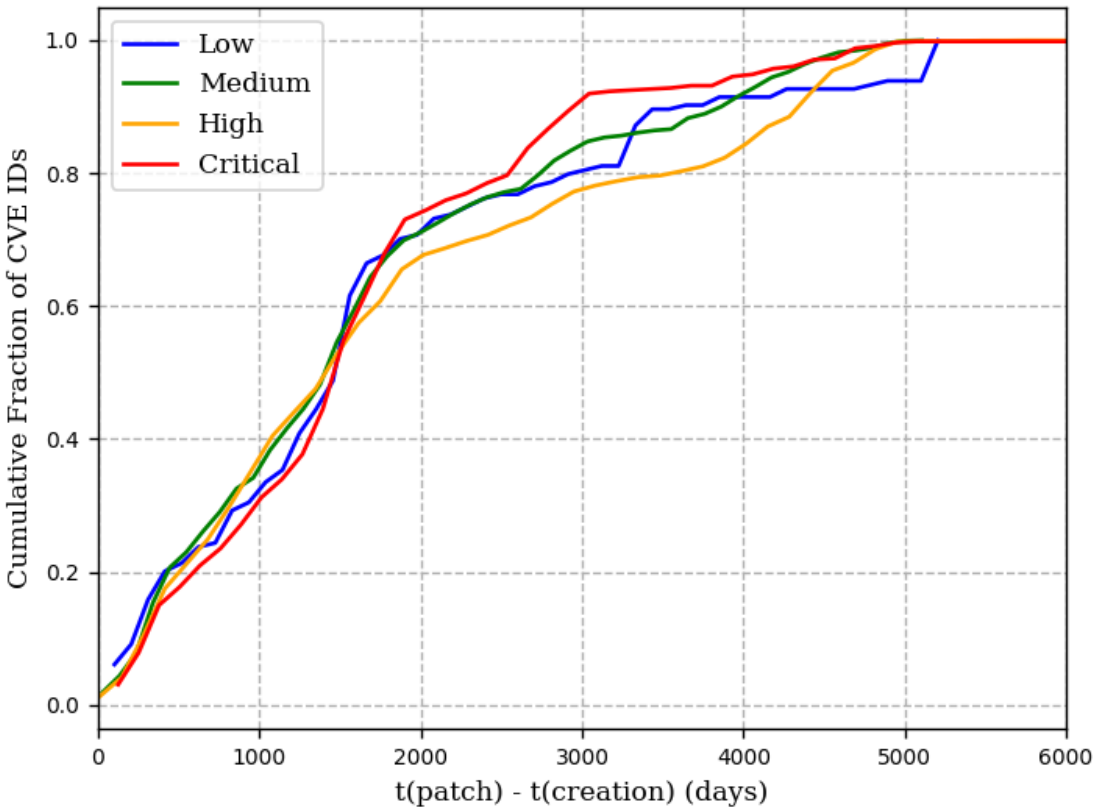

Figure 22. Longevity CDFs by CVSS severity (four-level)

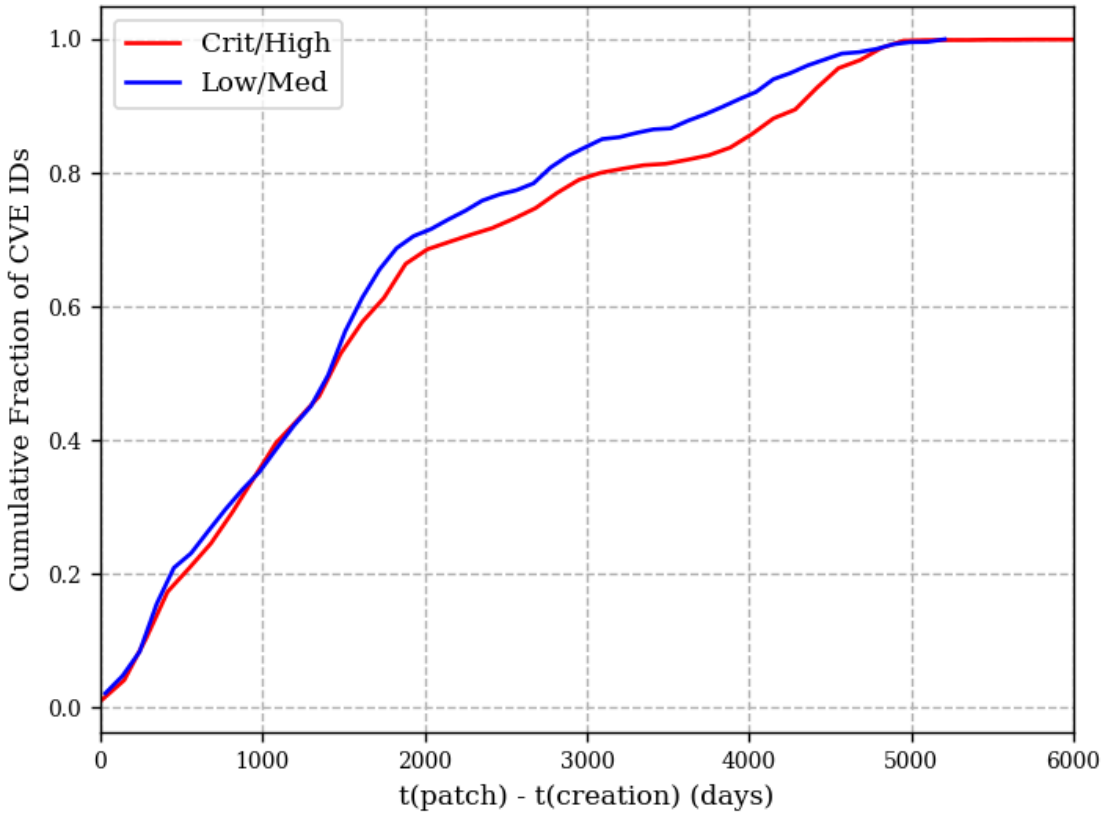

Figure 23. Longevity CDFs by CVSS severity (two-level)

When we compared patching behavior (Δ_{dp}) by CVSS severity, the statistics were like the overall longevity data: The datasets for each level had large standard deviations. The medians were all similar and close to zero, indicating that at least half of the vulnerabilities were patched on or before their disclosure. Figure 24 shows the CDFs for Δ_{dp} at each severity level; little difference existed between them except for Low-severity vulnerabilities, for which a greater portion appeared to be patched before disclosure before falling below the other levels during the first 200 days after disclosure. However, we should note that the size of that subset was small (233 vulnerabilities) compared to the others.

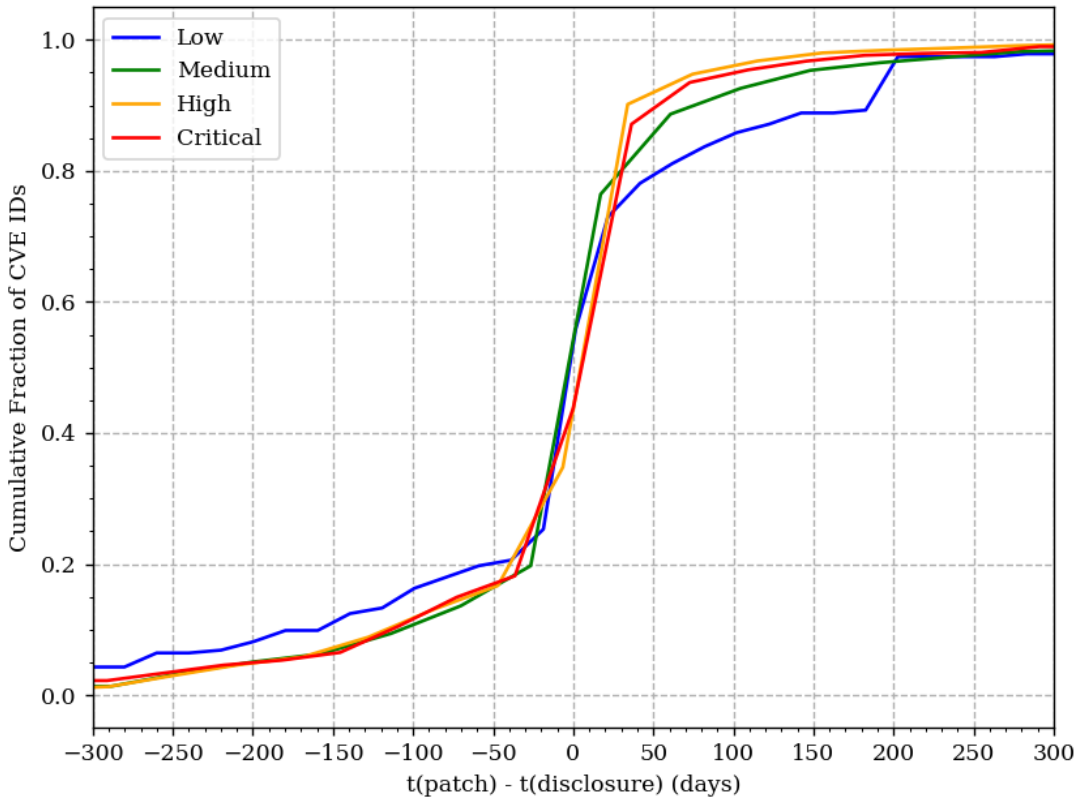

Figure 24. CDF of time to patch by CVSS severity

4. Operating System

We grouped our product lines into four groups for analysis: Windows, Linux distributions, Apple products, and Android (Figures 25–28). Google Chrome had too few operating system vulnerabilities for any meaningful analysis; this is probably because it is based on an application, so most Chrome vulnerabilities are not classified as operating system vulnerabilities, even if they affect ChromeOS. We found that Apple operating system vulnerabilities had the shortest longevity: The median Δ_{cp} was 737 days for Apple. Android was 909 days, Linux was 1,011 days, and Windows was over 10 years (3,654 days). Less than 40 percent of Windows vulnerabilities were patched after 2,500 days (6.8 years). In contrast, Apple and Android lacked any vulnerabilities whose lifespan exceeded 1,400 days.

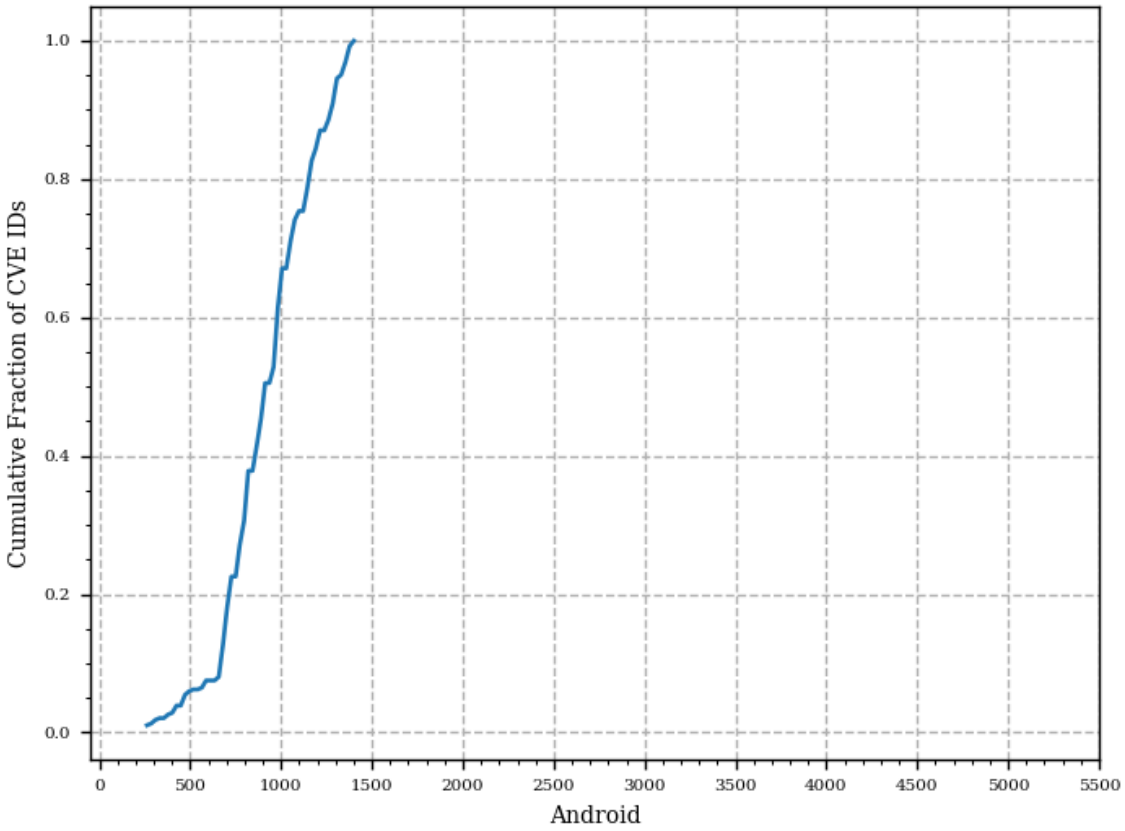

Figure 25. Longevity CDF – Android

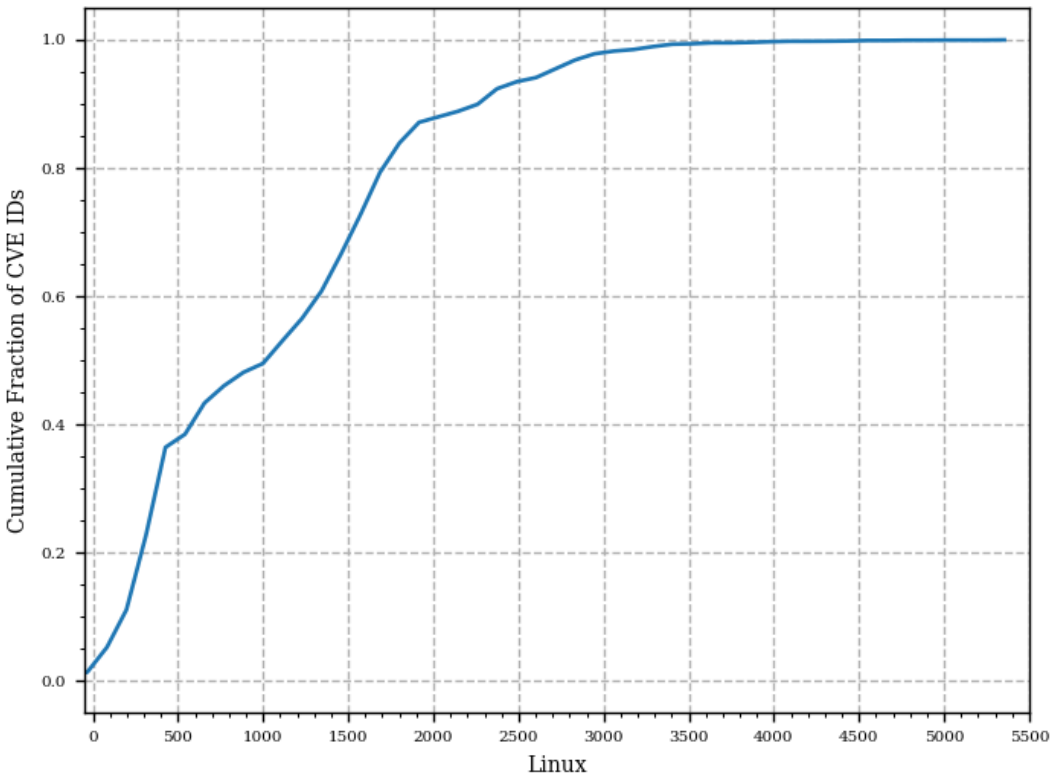

Figure 26. Longevity CDF – Linux distributions

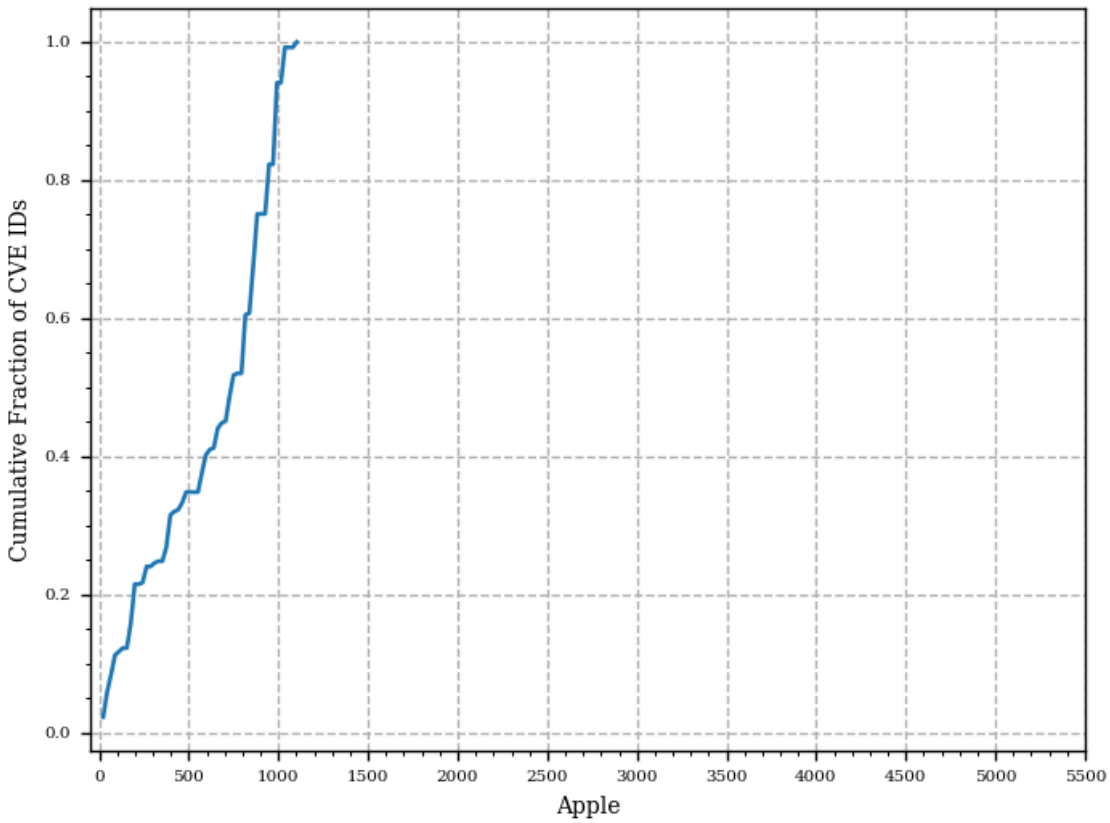

Figure 27. Longevity CDF – Apple

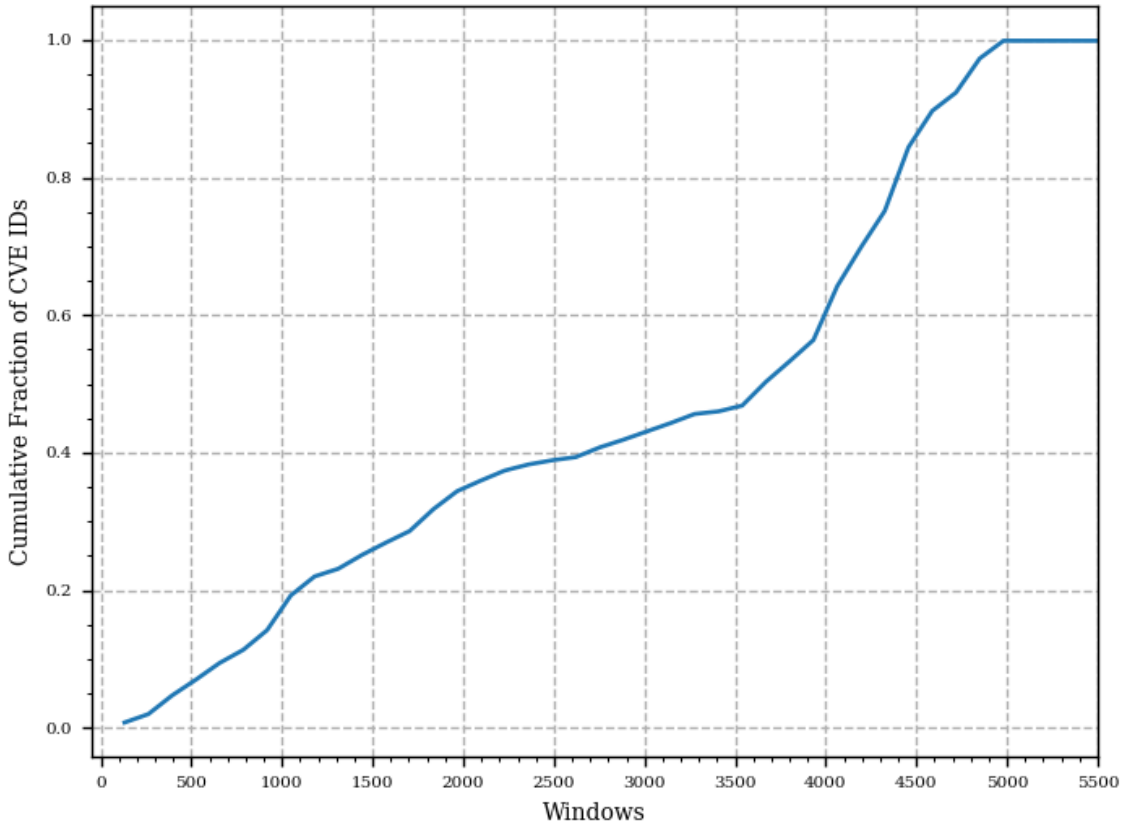

Figure 28. Longevity CDF – Windows

A possible reason that vulnerabilities for Apple and Android have shorter lifespans is that they release major operating system versions more often than their counterparts. For example, Apple's macOS 10.14 (Mojave) was released on September 24, 2018, but reached end-of-life in late 2021, a lifespan of only three years (Ng, 2022; Apple Inc., 2022). Vendors are unlikely to disclose vulnerabilities for products after they reach end-of-life because security support and software updates are no longer provided. Even if the vulnerability affects versions currently in use, a vendor is probably less likely to mention legacy versions when disclosing vulnerabilities.

We also studied whether variations in patching behavior existed between each operating system group; the CDFs for Δ_{dp} are shown in Figures 29–32. Apple, Android, and Windows each patched over 80 percent of their vulnerabilities before their disclosure date. What is noteworthy about Windows is that out of 2,138 vulnerabilities in the

Windows subset, all were patched on or before the date they were disclosed. In contrast, the Linux-based operating systems were much slower to patch their vulnerabilities. One reason could be that unlike with the other operating systems, Linux vendors rely heavily on community development and vulnerabilities must be disclosed to mobilize the developers to create a patch.

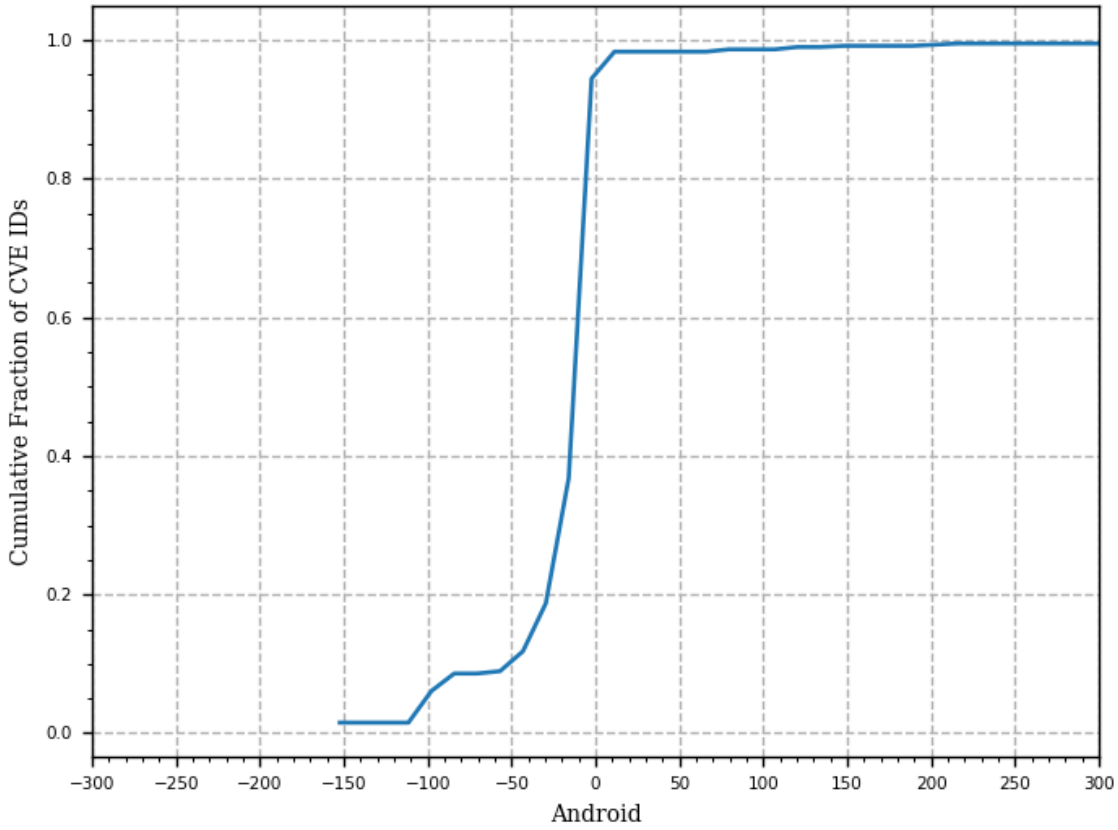

Figure 29. CDF for Δ_{dp} – Android

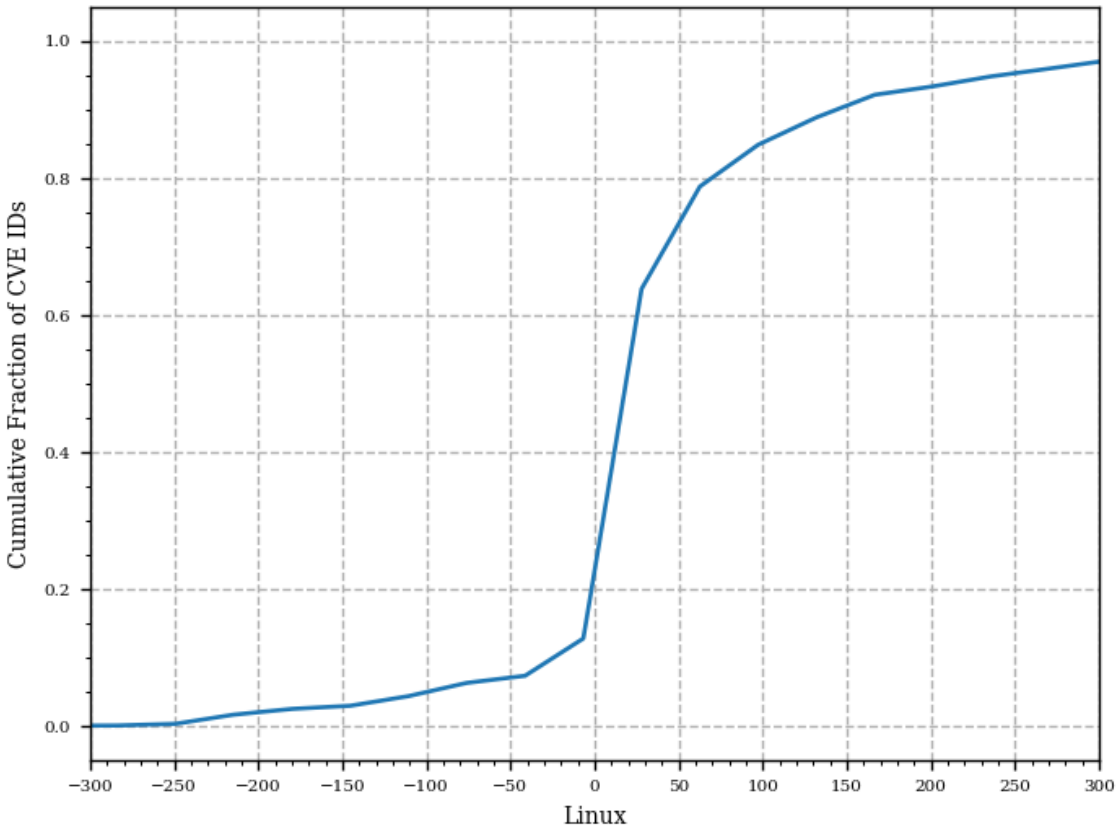

Figure 30. CDF for Δ_{dp} – Linux distributions

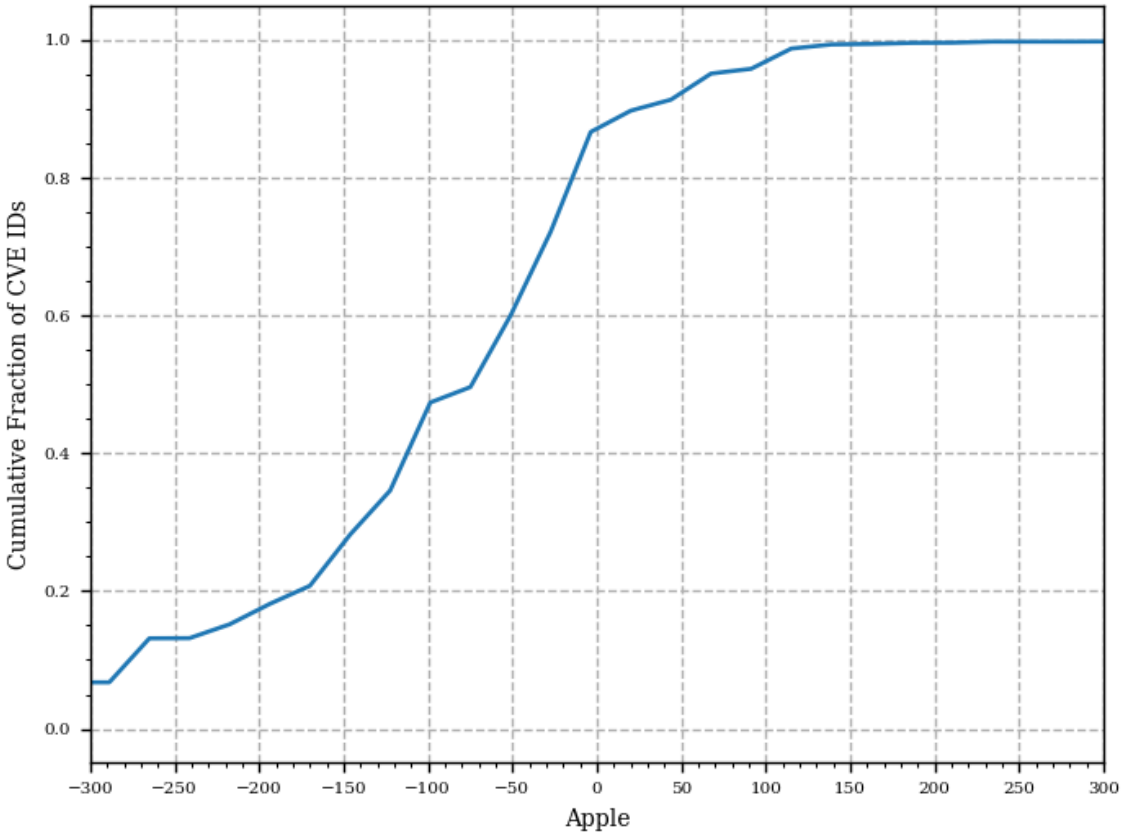

Figure 31. CDF for Δ_{dp} – Apple

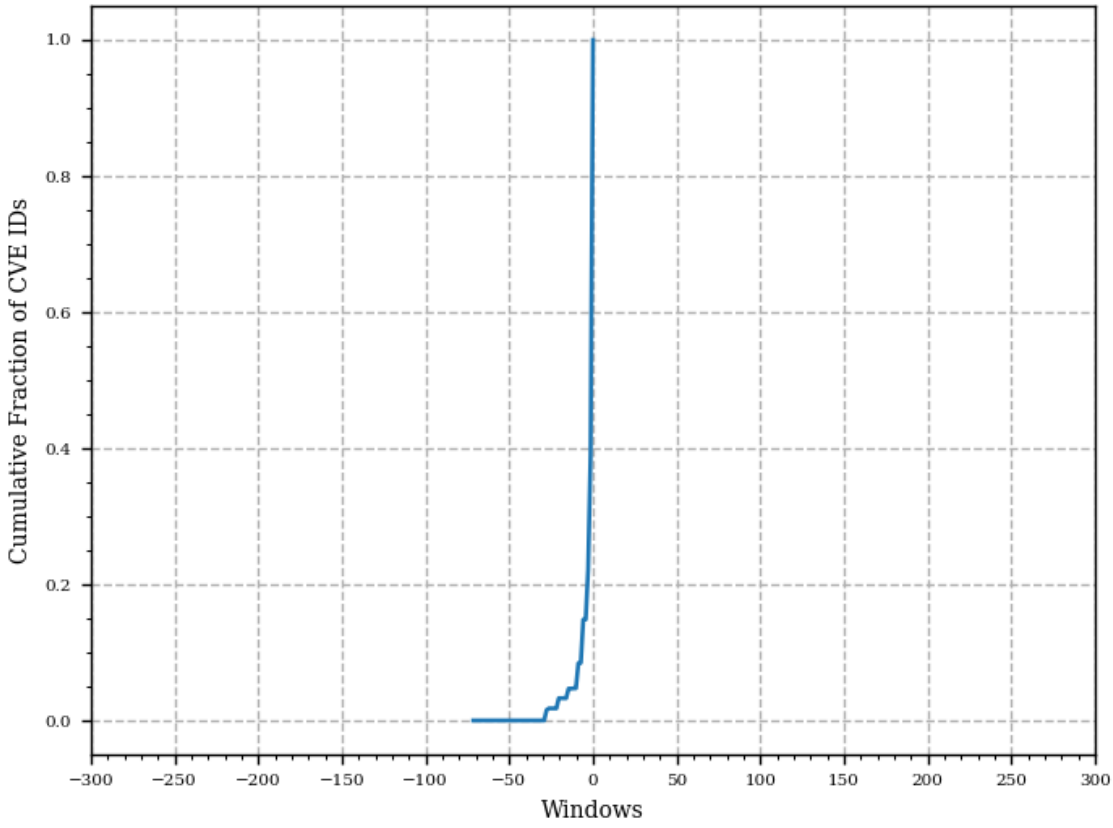

Figure 32. CDF for Δ_{dp} – Windows

VI. CONCLUSION

A. OVERVIEW

We successfully extracted dates for the creation, disclosure, patch, and exploitation of vulnerabilities associated with common operating systems. We used this data to calculate the lengths of software-vulnerability life cycle phases and plot survival functions for longevity and time to patch given that the patch was released after disclosure of the vulnerability. The median expected longevity for a vulnerability was 1,485 days (4.07 years), based on the Kaplan-Meier estimator, and 25.09 days after disclosure, based on the Weibull model.

Our median expected longevity was considerably shorter than previously found in work that measured the longevity of exploits (not vulnerabilities) and determined their median survival time to be 5.07 years and 6.9 years on average (Ablon & Bogart, 2017). This is noteworthy because the methods used in that study calculated longevity starting with the date of vulnerability discovery, not creation, and ending with public disclosure of the vulnerability. With parameters similar to those used in this thesis, the survival time would almost certainly be longer, and the difference between this thesis and previous work would be greater. This may be due to a lack of vendor data for legacy operating systems that have passed their end-of-life, which may make the vulnerability lifespans appear shorter.

We also studied the effects of vulnerability severity and variations between operating systems. While previous work found that vendors patched higher severity vulnerabilities more quickly (Shazad et al., 2020), our results did not show that more severe vulnerabilities were disclosed or patched more quickly than those of lower severity. Our finding that open-source vendors (the Linux-based distributions) patched their vulnerabilities more slowly than closed-source vendors was also consistent with this previous research. However, Microsoft's ability to patch all its Windows vulnerabilities before disclosure was a significant difference from prior findings. We also found that Windows and Linux vulnerabilities had higher longevity than Apple or Android. However,

this difference may be due to Apple and Android operating systems reaching end-of-life faster than Linux or Windows; the actual longevity could be longer if legacy versions were included.

Our assumption that exploits published before disclosure were used in malicious software and cyberweapons was incorrect. We neglected to consider that cybersecurity experts and researchers populating the Exploit Database may also collaborate privately with vendors. Most exploits developed before disclosure were indeed patched before the exploits were published, indicating such coordination. When we tried to correct for this by reducing the dataset to only those vulnerabilities whose exploit publishing date was earlier than the patch release date, too few vulnerabilities remained for us to glean any useful data. We extracted some useful vendor response data using a part of the dataset for Δ_{dp} that had positive values, although that method removed the effect of a known or discovered exploit, which could affect the time a vendor takes to release a patch.

This thesis also only studied operating-system vulnerabilities. Our dataset did not include application or firmware vulnerabilities from the NVD (NIST, 2011). These may have different characteristics affecting their lifespans that would be important for cyber operations planners to be aware of when selecting an appropriate cyberweapon. Also, vendors for applications or firmware may patch at a different speed than operating system vendors, which would affect cyberweapon reuse.

B. STRATEGIC IMPLICATIONS

With the survival function and associated Weibull distribution for longevity, we can determine the probability of a vulnerability's survival from the date it was first created. We can then assess what useful life it may have left. Using the survival function and associated Weibull distribution from time to patch after disclosure, we can also estimate the probability that a vendor will release a patch and thus assess the window for reuse.

While perishability and obsolescence create incentives to stockpile and use exploits, our work provides some insight about which exploits to stockpile and for how long, diminishing the cyberweapon arms race that results from worst-case scenario planning (Huntley, 2016). Instead of purchasing or developing as many cyberweapons as

possible, a better approach would be to develop cyberweapons exploiting vulnerabilities early in their life cycle, and then plan to develop or acquire replacements when the vulnerability reaches a specified probability threshold of patching or obsolescence. The low likelihood that a vulnerability will be re-discovered by others (Ablon & Bogart, 2017) lends support to this approach. This can reduce costs in money, manpower, and time, although a well-resourced offensive cyber actor may still choose to stockpile as many exploits as possible. Besides obsolescence, actors can use the probability of survival after disclosure to assess perishability and establish a timeframe and priority for subsequent attacks.

Determining the probability of survival of a vulnerability could create an incentive to use cyberweapons to exploit older vulnerabilities; this is a "use it or lose it" incentive associated with obsolescence (Huntley, 2016). While exploit development times have been observed to be relatively short (Ablon & Bogart, 2017), the process of finding new vulnerabilities or purchasing exploits from developers can be time-consuming and expensive (Smeets, 2018); Ablon and Bogart found that the typical market price for an exploit is in the $50,000 - $100,000 range. Therefore, poorly-resourced actors may have a greater incentive to use older exploits before they are patched because they may be unable to replace them.

Another factor that could encourage use of a cyberweapon is whether an actor knows that a vulnerability will be patched soon. If most vulnerabilities are patched before disclosure, then the release of a patch is probably the first indication that a cyberweapon is becoming ineffective. Disclosure before patch release still offers a small window for a cyberweapon to be used, permitting some potential return on investment from its development. Then there would be a strong incentive to use a cyberweapon once it has been disclosed. If a vulnerability is not disclosed, the loss of that window encourages using a cyberweapon earlier in its life cycle. It is worth noting that some vendors are better at patching before disclosure than others. We found that Windows patched all its vulnerabilities on or before disclosure; in contrast, Linux vendors patched very few before disclosure, probably due to their open-source development model.

The wide variability in a vulnerability's longevity (as indicated by large standard deviations) suggests that risk tolerance is a major factor when deciding when to use or replace a cyberweapon. Had most of the longevity values been clustered around the mean with smaller standard deviations, a long window of relatively low risk that a patch would be released would occur, followed by a short high-risk period. With a wider dispersion of longevity values, the risk is distributed over a longer duration, and this could mean a difference of several years. If risk tolerance dictates that cyberweapons be used or replaced before their vulnerabilities reach a 75 percent probability of survival, then the window of opportunity is 775 days from creation using our Weibull model. If actors can wait until there is a 25 percent probability, then the window of opportunity is 2,639 days. This would have significant implications not just for operations, but also for staffing and funding of new cyberweapon development and acquisition.

C. APPLICATION TO CYBERSPACE OPERATIONS

The Weibull distribution model fitted to our longevity data yielded a distribution of the probability that a vulnerability will survive past a specific number of days after its creation. This enables planners to predict a vulnerability's obsolescence. Longevity was our primary metric for obsolescence because most vulnerabilities in our dataset were patched on or before the date they were publicly disclosed. This suggests that vendors are actively collaborating with and incentivizing white-hat and other security professionals to disclose vulnerabilities privately to them, then wait until a patch is released to disclose it publicly.

For cyberspace-operation planners, this means they can assess the remaining useful life of a cyberweapon based on the probability associated with the survival function. This would be valuable when choosing cyberweapons, as a cyberweapon that exploits an older vulnerability that has less than a 50 percent probability of survival may be less desirable than one whose exploited vulnerability is fewer days removed from the date it was created. This is also useful for determining whether to develop a cyberweapon: If operators find

that the survival function for an older cyberweapon shows a low probability of survival, they can prioritize development of new cyberweapons with similar effects. As mentioned earlier, risk tolerance is also a factor; lower risk tolerance means that planners would use cyberweapons earlier in their vulnerability's life cycle and replace them more frequently.

While a vulnerability's severity did not seem to affect its life cycle, the operating system did. For example, a vulnerability in Apple operating systems will probably have a shorter lifespan than a vulnerability in Windows, so a cyber operations planner should use a cyberweapon targeting an Apple operating system earlier than they would a cyberweapon targeting Windows, and replacement cyberweapons should also be developed more often. Planners should also consider the vendor's propensity to patch vulnerabilities before disclosure. Windows, Apple, and Android had many vulnerabilities patched before they were disclosed, but Linux did not. Therefore, there is less risk of obsolescence with a cyberweapon that targets Linux operating systems, as there will probably be more of a window to use it before a patch is released.

However, the probability of survival is just one factor in selecting cyberweapons. Planners should also consider whether the target software and operating system are still receiving security support, or the likelihood the vendor would patch legacy software. We could not determine if the presence of a known exploit would affect the time it took a vendor to release a patch. However, we could develop a survival model for vulnerabilities where $\Delta_{dp} > 0$, which we assumed were vulnerabilities the vendor was unaware of before disclosure. The survival function in Figure 13 permits this. Planners can then determine the period within which they can redeploy that weapon after initial use, prioritizing their targets to maximize operational effect. For example, a planner may schedule subsequent cyberweapon use against high-value targets within the first 25 days after it is first deployed when the probability of survival is greater than 50 percent. After that point, when the risk of mission failure is greater, lower-priority targets or targets of opportunity could then be scheduled. Before any such operation, careful planning could determine what additional targets could be attacked, what priority is assigned to each, and whether hastening attack on these targets is consistent with operational objectives; there will inherently be trade-offs (Smeets, 2018).

D. FUTURE WORK

Future work could measure the software-vulnerability phases of application or firmware vulnerabilities because they are also workable vectors for exploits that cyber operations planners could use. Determining the probability of survival for vulnerabilities that have been exploited also warrants further study due to its potential effect on cyberweapon reuse. Future work could also explore characteristics of different types of vulnerabilities that affect their longevity or the phases of their life cycles.

APPENDIX A. PYTHON SCRIPT FOR PARSING JSON FILES

A. PROGRAM DESCRIPTION

The Python program below (*JSONParse.py*) loads the specified set of JSON files downloaded from the NVD. It then converts them to Python nested dictionaries and parses them to extract the required data for each CVE entry, including the CVE ID, CVSS score, publishing date, and the affected software versions for each vendor and product line. If the CVE ID is within our desired dataset, it is added to the CVE Index. When this process is complete, the CVE Index is exported to a CSV file. Also, the index is broken up by product line, then exported separately to assess metrics for each operating system. Lastly, the complete set of affected software versions is exported to CSV files, one for each product line, which are used for populating software version release dates.

B. SOFTWARE CODE

```
#############################################################################
# Program Name: JSONParse.py
# Author: Michael Lidestri
# Date of Publication: May 16, 2022
#
# Description: JSONParse.py opens four JSON files (years 2018–2021) from the NVD and
# converts the contents to a Python nested dictionary. This dictionary is then used to
# extract the vendor, product, version, publishing date and other required data. This
# information is then inserted into a CVE index, cve_dict, which holds all CVEs. It also
# generates the product/version information needed to manually search for creation dates.
#############################################################################

import json
import csv
from datetime import date, datetime
import dateutil.parser as dparser

######################## GLOBAL VARIABLES ############################
# These variables are used to parse, store, and export CVE information
cve_dict= {}        # CVE Index where all CVEs and their data will be stored
os_list = ['windows', 'chrome', 'iphone', 'macos', 'mac', 'enterprise', 'fedora', \
        'android', 'tvos', 'watchos', 'debian', 'linux', 'ubuntu']
vendor_list = ['canonical', 'microsoft', 'google', 'redhat', 'apple', 'linux',\
          'fedoraproject', 'debian']
```

```
# These variables are used to generate statistics about the dataset
vers_master = {}
vendor_master = set()
vendor_dict = {}    #
sw_dict = {}
sw_master = set()
no_cvss = 0
count = 0
###############################################################################

def getVersion(vers_str):
  # Determines if the version string contains version data or just filler characters.
  # It returns a version only if there are alphanumberic characters and None otherwise.
  res = any(chr.isalnum() for chr in vers_str)
  if res:
     min_vers = vers_str
  else:
     min_vers = None

  return min_vers

def getData(cpe):
  # This function splits a CPE string into its components: part, vendor, affected
  # software (product line), product, and version. See (NIST, 2011) for more details
  # about CPE strings.
  global vendor_master
  global sw_master

  cpe_list = cpe.split(':')
  part = cpe_list [2]
  vendor = cpe_list [3]
  sw_aff = cpe_list [4].split('_')
  vers_string = cpe_list [5]
  product = sw_aff [1:]
  new_sw = sw_aff [0]
  sw_master.add(new_sw)
  vendor_master.add(vendor)

  return part, vendor, new_sw, product, vers_string

def processJSON(data):
  # Takes a JSON file (converted to Python nested dictionaries) and parses it. It
  # extracts the CVE, CVSS Score, affected software product line, and version. If the
  # data is part of the corpus we intend to use for research, it is placed in the CVE
```

```
# Index (cve_dict).
global count
global cve_dict
global no_cvss
global other_count

for i in data ['CVE_Items']:
    # Extract CVE, CPE data; create data structures for affected software
    cve = (i ['cve']['CVE_data_meta']['ID'])
    configs = i ['configurations']['nodes']
    cpe_list = []
    vendor_set = set()
    sw_set = set()
    vers_dict = {}
    in_corpus = False
    in_sw = False

    # Extract the publishing date, which will be t(disclosure), and CVSS score
    publishedDate = i ['publishedDate'][:10]
    try:
        cvss_score = i ['impact']['baseMetricV3']['cvssV3']['baseScore']
    except:
        cvss_score = 'N/A'
        no_cvss +=1

    # Extract product line and version data
    for node in configs:
        # if each node has a 'children' field, the following code runs data extraction
        for index in node ['children']:
            for number in index ['cpe_match']:
                if number ['vulnerable'] == True:
                    cpe = number ['cpe23Uri']  # This is the CPE string
                    # Extract data from the CPE string
                    part, vendor, new_sw, product, vers_string = getData(cpe)
                    # If product is an operating system, extract software+version data
                    if part == 'o':
                        vendor_set.add(vendor)
                        sw_set.add(new_sw)

                        if new_sw in os_list and vendor in vendor_list:
                            in_corpus = True
                        if new_sw in os_list:
                            in_sw= True

                        # Check for varying sources of version data
```

```
            if ‘versionStartExcluding’ in number:
                version = str(‘>‘+number [‘versionStartExcluding’])
            elif ‘versionStartIncluding’ in number:
                version = number [‘versionStartIncluding’]
            else:
                version = getVersion(vers_string)

            if version != None:
                product.append(version)
            prod_vers = ‘.’.join(product)

            if new_sw not in vers_dict:
                vers_dict [new_sw] = set()
            vers_dict [new_sw].add(prod_vers)

# if a node does not have a ‘children’ field, data extraction starts here
for index in node [‘cpe_match’]:
    if index [‘vulnerable’] == True:
        cpe = index [‘cpe23Uri’]
        # Extract data from the CPE string
        part, vendor, new_sw, product, vers_string = getData(cpe)
        # If product is an operating system, extract software and version data
        if part == ‘o’:
            vendor_set.add(vendor)
            sw_set.add(new_sw)
            if new_sw in os_list and vendor in vendor_list:
                in_corpus = True
            if new_sw in os_list:
                in_sw= True

            if ‘versionStartExcluding’ in index:
                version = str(‘>‘+index [‘versionStartExcluding’])
            elif ‘versionStartIncluding’ in index:
                version = index [‘versionStartIncluding’]
            else:
                version = getVersion(vers_string)

            if version != None:
                product.append(version)
            prod_vers = ‘.’.join(product)

            if new_sw not in vers_dict:
                vers_dict [new_sw] = set()
            vers_dict [new_sw].add(prod_vers)
```

```
    # If part of our dataset, add to the CVE Index, if not already listed
    if in_corpus:
       if cve not in cve_dict:
          cve_dict [cve] = {'CVSS Score' : cvss_score, 't(creation)' : 'N/A',\
                    't(disclosure)': publishedDate, 't(patch)' : 'N/A', \
                    't(exploit)' : 'N/A', 'Vendor' : vendor_set, \
                    'Affected Software' : sw_set, 'Versions' : vers_dict}

    count+=1
############################ MAIN FUNCTION ##############################

if __name__ == "__main__":
  # Imports JSON files specified below from NVD (NIST, 2022) and converts them to
  # Python dictionaries, which are then processed so data can be extracted and inserted
  # into the CVE Index. When data has been extracted from all files, the CVE Index and
  # compiled software/version data are exported to CSV files for further processing.

  # Load JSON files and process (Canepa, 2019).
  json_file_list = ('nvdcve-1.1-2021.json', 'nvdcve-1.1-2020.json', \
              'nvdcve-1.1-2019.json', 'nvdcve-1.1-2018.json')

  for file in json_file_list:
     json_file = open(file, encoding='utf8')
     json_data = json.load(json_file)
     processJSON(json_data)
     json_file.close()

  #  Write CVE Index to CSV file (GeeksforGeeks, 2022).
  with open('cve_prelim_results.csv', 'w') as csvfile:
     fields = ['cve', 'cvss score', 'vendor', 'affected software', 'versions', \
           't(creation)', 't(disclosure)', 't(patch)', 't(exploit)']
     writer = csv.DictWriter(csvfile, fieldnames = fields)
     writer.writeheader()

     for item in cve_dict:
        line = {'cve': item, 'vendor' : cve_dict [item]['Vendor'], \
              'affected software' : cve_dict [item]['Affected Software'], \
              'versions' : cve_dict [item]['Versions'], 'cvss score' : \
              cve_dict [item]['CVSS Score'], 't(creation)': \
              cve_dict [item]['t(creation)'], 't(disclosure)' : \
              cve_dict [item]['t(disclosure)'], 't(patch)': \
              cve_dict [item]['t(patch)'], 't(exploit)': cve_dict [item]['t(exploit)']}

        writer.writerow(line)
```

```
# Break out CVE Index by operating system for product specific calculations later,
# export to CSV file (GeeksforGeeks, 2022).
for os in os_list:
    with open(str(os)+'_prelim_results.csv', 'w') as file:
        fields = ['cve', 'cvss score', 'vendor', 'affected software', 'versions', \
                  't(creation)', 't(disclosure)', 't(patch)', 't(exploit)']
        writer = csv.DictWriter(file, fieldnames = fields)
        writer.writeheader()

        for item in cve_dict:
            if os in cve_dict [item]['Affected Software']:
                line = {'cve': item, 'vendor' : cve_dict [item]['Vendor'], \
                        'affected software' : cve_dict [item]['Affected Software'], \
                        'versions' : cve_dict [item]['Versions'], 'cvss score' : \
                        cve_dict [item]['CVSS Score'], 't(creation)': \
                        cve_dict [item]['t(creation)'], 't(disclosure)' : \
                        cve_dict [item]['t(disclosure)'], 't(patch)': \
                        cve_dict [item]['t(patch)'], 't(exploit)': \
                        cve_dict [item]['t(exploit)']}
                writer.writerow(line)

######################## GET PRODUCT/VERSION DATA ####################
# This portion of the main function extracts vendor, software, and version data; it
# then exports the data to individual CSV files for each vendor. It then calculates
# and prints some statistics for the dataset.
vendor_count = 0
multi = 0
corpus_count = 0

# Generate totals for number of vendors, products, versions
for item in cve_dict:
    counted = False
    vendors = cve_dict [item]['Vendor']
    software = cve_dict [item]['Affected Software']
    versions = cve_dict [item]['Versions']
    if len(vendors) > 1:
        multi +=1
    for vendor in vendors:
        if vendor not in vendor_dict:
            vendor_dict [vendor] = 0
        vendor_dict [vendor] += 1
        vendor_count += 1

    for sw in software:
        if sw not in sw_dict:
```

```
            sw_dict [sw] = 0
        sw_dict [sw] += 1
        if sw in os_list and counted == False:
            corpus_count += 1
            counted = True

    for program in versions:
        if program not in vers_master:
            vers_master [program] = set()
        for version in versions [program]:
            vers_master [program].add(version)

# For each operating system, extract the set of products/versions associated with CVEs
# in the dataset, and export them to a CSV file for further processing
# (GeeksforGeeks, 2022).
for index in os_list:
    rows = []
    if index in vers_master:
        with open(index+'.csv', 'w') as file:
            for unit in vers_master [index]:
                row = [unit]
                rows.append(row)
            write = csv.writer(file)
            write.writerows(rows)
        file.close()

# Print statistics for dataset
print('Number of CVEs: ', count)
print('Number of Vendors: ', vendor_count)
print('CVEs Affecting more than one vendor:', multi)
print('Number of Unique Vendors: ', len(vendor_master))
print('Number of Unique Software: ', len(sw_master))
print('Corpus Count: ', corpus_count)
print('CVEs with no CVSS: ', no_cvss)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SCRAPY SPIDER EXAMPLE

A. PROGRAM DESCRIPTION

The Python program below is an example of a spider we used to scrape patch and exploit dates for CVE IDs. The spiders rely on central files generated by the Scrapy module when the crawler is created. While each spider is unique, each one loads the HTML page specified in the start URL, parses it for links to follow to security bulletins, and loads those pages, from which it extracts CVE IDs and publishing dates.

B. SOFTWARE CODE

```
##############################################################################
# Program Name: ubuntu_patches.py
# Author: Michael Lidestri
# Date of Publication: May 16, 2022
#
# This program is a spider based off the Scrapy module (Zyte, 2022). The spider class was
# generated by Scrapy and leverages settings and middleware files that are also generated
# by the Scrapy package. We customized the spider to crawl the desired webpages and
# extract the data we needed. (Jabeen, 2019) is an excellent tutorial we used to learn how
# to create a webcrawler and spiders in Scrapy. For XPath guidance, see (Scrapy
# Developers, 2022), which we used to build our queries.
#
# Description: Starting from the start URL, retrieves the HTML page, parses it for links to
# security bulletins, then creates new requests to retrieve those HTML pages, with a call
# to the getCVE() function which extracts the patch date and CVE ID. Yields a CVE ID
# and  patch date, which are exported to the CSV file specified under custom_settings.
##############################################################################

from datetime import date, datetime
import dateutil.parser as dparser
from ..items import PatchItem

class UbuntuPatchesSpider(scrapy.Spider):
  name = 'ubuntu_patches'
  allowed_domains = ['ubuntu.com/']
  start_urls = ['https://ubuntu.com/security/notices/']
  custom_settings= { 'FEED_URI': "ubuntu_patches.csv"}

  def parse(self, response):
     # Parses an HTML file; in this case it is the index page with links to security
```

```
    # bulletins. Retrieves all links and creates new GET messages to retrieve pages. Calls
    # getCVE() function for all subsequent pages.
    min_date = date(2018, 1, 1)

    # If website articles have dates within the cutoff for the dataset, yield a HTTP
    # request and pass to getCVE() to extract data. Dates extracted using dateutil
    # package (Niemeyer, 2019).
    posts = response.xpath("//div [contains(@class, 'col-9')]/article")
    for post in posts:
        post_date = post.xpath('p [1]/text()').get()
        extracted_date = dparser.parse(post_date, fuzzy=True)
        ext_date_adj = date(extracted_date.year, extracted_date.month,\
                        extracted_date.day)

        if ext_date_adj >= min_date:
            link = post.xpath("h3/a/@href").get()
            yield response.follow(url=link, callback=self.getCVE, dont_filter=True)

    if ext_date_adj >= min_date:
        url = response.xpath("//div/ol/li/a [contains(@class, \
        'p-pagination__link--next')]/@href").get()
        yield scrapy.Request(response.urljoin(url), callback=self.parse,\
                        dont_filter=True)

def getCVE(self, response):
    # Parses an HTML file to extract the CVE and the patch publishing date. Yields a
    # dictionary consisting of CVE ID and patch date. Ensures only stable updates are
    # used.
    prefix = "CVE-"

    # Extract patch date with dateutil package (Niemeyer, 2019)
    rel_date= response.xpath("//div [contains(@class, 'col-12')]/p [1]/text()").get()
    ext_date = dparser.parse(rel_date, fuzzy=True)
    new_ext_date = date(ext_date.year, ext_date.month, ext_date.day)
    formatted_date = new_ext_date.isoformat()

    # Extract CVE IDs and export with patch date to CSV file
    div_entries= response.xpath("//div [contains(@class, 'col-8')]/\
    ul [contains(@class, 'p-list')]/li [contains(@class, 'p-list__item')]/a/text()").getall()
    for entry in div_entries:
        item = PatchItem()
        if prefix in entry:
            item ['cve'] = entry
            item ['pdate'] = formatted_date
            yield(item)
```

LIST OF REFERENCES

3i Data Scraping. (2021, September 8). *How Scrapy and Selenium is used in analyzing and scraping news articles?* https://www.3idatascraping.com/how-scrapy-and-selenium-is-used-in-analyzing-and-scraping-news-articles.php

Ablon, L., & Bogart, A. (2017). *Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits* (Report No. RR1751). RAND. https://www.rand.org/pubs/research reports/RR1751.html

Anderson, T. (2020, January 6). *Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd*. The Register. https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/

Apple, Inc. (2022). *Apple security updates (2018 to 2019)*. https://support.apple.com/en-us/HT213078

Arbaugh, W., Fithen, W., & McHugh, J. (2000). Windows of vulnerability: A case study analysis. *Computer*, *33*(12), 52–59. http://dx.doi.org/10.1109/2.889093

Arora, A., Krishnan, R., Telang, R., & Yang, Y. (2010). An empirical analysis of software vendors' patch release behavior: Impact of vulnerability disclosure. *Information Systems Research*, *21*(1), 115–132. http://www.jstor.org/stable/23015522

Bronk, C., & Tikk-Ringas, E. (2013). The cyber attack on Saudi Aramco. *Survival: Global Politics and Strategy, 55*(2), 81–96. https://doi.org/10.1080/00396338.2013.784468

Canepa, G. (2019, January 15). *Importing data from a JSON resource with Python*. PluralSight. https://www.pluralsight.com/guides/importing-data-from-json-resource-with-python

Caswell, T., Droettboom, M., Lee, A., Andrade E., Hoffmann, T., Hunter, J., Klymak, J., Firing, E., Stansby, D., Varoquaux, N., Hedegaard Nielsen, J., Root, B., May, R., Elson, P., Seppänen, J., Dale, D., Lee, J., McDougall, D., Straw, A., Ivanov, P. (2021). matplotlib/matplotlib: REL: v3.5.1 (v3.5.1) [computer software]. Zenodo. https://doi.org/10.5281/zenodo.5773480

Clark, S.; Frei, S.; Blaze, M.; & Smith, J. (2010). Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. *Proceedings of the 26th Annual Computer Security Applications Conference* (*ACSAC '10), 251–260*. https://doi.org/10.1145/1920261.1920299

Cybersecurity and Infrastructure Security Agency. (2018, February 15). *Alert (TA17-181A)-Petya ransomware*. https://www.cisa.gov/uscert/ncas/alerts/TA17-181A

Dacey, R. (2003). *Information security effective patch management is critical to mitigating software vulnerabilitie*s (GAO-03-1138T). Government Accountability Office. https://www.gao.gov/assets/gao-03-1138t.pdf

Davidson-Pilon, (2019). Lifelines: survival analysis in Python. *Journal of Open Source Software*, 4(40), 1317, https://doi.org/10.21105/joss.01317

Elisan, C. (2015). *Advanced Malware Analysis*. McGraw-Hill.

Federal Bureau of Investigation. (2020, August 03). *Computer network infrastructure vulnerable to Windows 7 end of life status, increasing potential for cyber attacks.* https://www.documentcloud.org/documents/ 7013545-Windows-7-End-of-Life-PIN-20200803-002-BC.html

Frei, S., May, M., Fiedler, U., & Plattner, B. (2006). Large-scale vulnerability analysis. *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense* (*LSAD '06*), *131–138*. https://doi.org/10.1145/1162666.1162671

Frei, S., Dübendorfer, T., & Plattner, B. (2009). Firefox (in)security update dynamics exposed. *ACM SIGCOMM Computer Communication Review*, *39*(1), 16–22. https://doi.org/10.1145/1496091.1496094

Frei, S., Schatzmann, D., Plattner, B., & Trammell, B. (2010). Modeling the security ecosystem - The dynamics of (in)security. In T. Moore, D. Pym, and C. Ioannidis, (Eds.), *Economics of information security and privacy* (pp. 69–106). Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-6967-5_6

GeeksforGeeks. (2022, 19 February). *Working with csv files in Python*. https://www.geeksforgeeks.org/reading-csv-files-in-python/

Hall, C. (2017). *Time Sensitivity in Cyberweapon Reusability* [Master's thesis, Naval Postgraduate School]. NPS Archive, Calhoun. https://nps.primo.exlibrisgroup.com/permalink/01NPS_INST/ofs26a/alma991005653428203791

Harris, C. (2019). Agencies need to develop modernization plans for critical legacy systems (GAO-19-471). Government Accountability Office.

Herr, T., & Rosenzweig, P. (2014). Cyber weapons and export control: Incorporating dual use with the PrEP model. *Journal of National Security Law and Policy*, *8*(2), 301–319. http://dx.doi.org/10.2139/ssrn.2501789

Huntley, W. (2016). Strategic implications of offense and defense in cyberwar. *2016 49th Hawaii International Conference on System Sciences, 5588–5595.* http://dx.doi.org/10.1109/HICSS.2016.691

Hutchins, E., Cloppert, M., & Amin, R. (2011). *Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains*. Lockheed Martin Corporation. https://lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf

Ismiguzel, I. (2021, September 27). *Hands-on survival analysis with Python*. TOPBOTS. https://www.topbots.com/survival-analysis-with-python/

Jabeen, H. (2019, January 10). *Making Web crawlers using Scrapy for Python*. Datacamp. https://www.datacamp.com/tutorial/making-web-crawlers-scrapy-python

Johnson, T. (2018). Growing cybersecurity concerns in industrial IoT. *Telecom Asia (Online)*. https://libproxy.nps.edu/login?url=https://www.proquest.com/trade-journals/growing-cybersecurity-concerns-industrial-iot/docview/2139264772/se-2?accountid=12702

Kang, M., Pongsin, P., Yin, H. (2007). Renovo: a hidden code extractor for packed executables. *2007 ACM workshop on Recurring Malcode (WORM '07), 46–53.* https://doi.org/10.1145/1314389.1314399

Kleinbaum, D. & Klein, M. (2005). *Survival analysis: A self-learning text* (2nd ed.). Springer.

Kovacs, E. (2016, April 7). *OSVDB shut down permanently*. Security Week. https://www.securityweek.com/osvdb-shut-down-permanently

Langner, R. (2013). *To kill a centrifuge: A technical analysis of what Stuxnet's creators tried to achieve.* The Langner Group. https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf

Lee, R., Assante, M., Conway, T. (2016). *Analysis of the cyber attack on the Ukrainian power grid*. E-ISAC, SANS ICS. https://www.readkong.com/page/analysis-of-the-cyber-attack-on-the-ukrainian-power-grid-6826988

Lewinson, E. (2020, August 17). *Introduction to survival analysis: the Kaplan-Meier estimator*. Towards Data Science. https://towardsdatascience.com/introduction-to-survival-analysis-the-kaplan-meier-estimator-94ec5812a97a

Libicki, M., Ablon, L., Webb, A. (2015). *The defender's dilemma: Charting a course toward cybersecurity* (Report No. RR1024). RAND. https://www.rand.org/content/dam/rand/pubs/research_reports/RR1000/RR1024/RAND_RR1024.pdf

Liles, S., & Poremski, E. (2015). Fusion of malware and weapons taxonomies for analysis. *Journal of Information Warfare*, *14*(1), 75–83. https://www.jstor.org/stable/26487520

Mamka, A. (2016, August 29). *How to solve 403 error in scrapy*. Stack Overflow. https://stackoverflow.com/questions/39202058/how-to-solve-403-error-in-scrapy

Martelle, M. (2018, August 13). *Joint Task Force ARES and Operation GLOWING SYMPHONY: Cyber Command's Internet war against ISIL*. George Washington University. https://nsarchive.gwu.edu/briefing-book/cyber-vault/2018-08-13/joint-task-force-ares-operation-glowing-symphony-cyber-commands-internet-war-against-isil

Mele, S. (2014). Legal considerations on cyber-weapons and their definition. *Journal of Law & Cyber Warfare*, *3*(1), 52–69. http://www.jstor.org/stable/26432559

Metrick; K., Semrau, J., & Sadayappan, S. (2020, April 13). Think fast: Time between disclosure, patch release and vulnerability exploitation — Intelligence for vulnerability management, part two. *Mandiant*. https://www.mandiant.com/resources/time-between-disclosure-patch-release-and-vulnerability-exploitation

Microsoft. (2013). *Microsoft security intelligence report, volume 15*. Microsoft Corporation. https://www.microsoft.com/en-us/download/details.aspx?id=40871

Microsoft. (2022). Microsoft bug bounty program. Microsoft Corporation. https://www.microsoft.com/en-us/msrc/bounty

Nappa, A., Johnson, R., Bilge, L., Caballero, J., & Dumitras, T. (2015). The attack of the clones: A study of the impact of shared code on vulnerability patching. *2015 IEEE Symposium on Security and Privacy, 692–708*. https://doi.org/10.1109/SP.2015.48

National Institute of Standards and Technology. (2011). *Common product enumeration: Naming Specification Version 2.3* (IR 7695). https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7696.pdf

National Institute of Standards and Technology. (2022). *The National Vulnerability Database* [Data set]. https://nvd.nist.gov/vuln/data-feeds

Niemeyer, G. (2019). *Dateutil* (version 2.8.2) [computer software]. https://dateutil.readthedocs.io/en/stable/

Ng, D. (2022, March 7). *macOS 10.14 Mojave - end of support*. University of California – San Francisco. https://it.ucsf.edu/news/macos-1014-mojave-end-support

Offensive Security. (2022). *The Exploit Database* [Data set]. https://www.exploit-db.com/

Pandas Development Team. (2020). pandas-dev/pandas: Pandas 1.0.3 (v1.0.3) [computer software]. Zenodo. https://doi.org/10.5281/zenodo.3715232

Reliasoft. (2002, April). Characteristics of the Weibull Distribution. *Reliability Hotwire (14)*. https://weibull.com/hotwire/issue14/relbasics14.htm

Richardson, L. (2015). Beautiful Soup [computer software]. Leonard Richardson. https://beautiful-soup-4.readthedocs.io/en/latest/#

Sarabi, A., Zhu, Z., Xiao, C., Liu, M., & Dumitraş, T. (2017). Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state. In *Passive and Active Measurement* (pp. 113–125). Springer International Publishing. https://doi.org/10.1007/978-3-319-54328-4_9

Scrapy Developers. (2022, April 14). Scrapy 2.6 documentation. Scrapy. https://docs.scrapy.org/en/latest/

The Selenium Project. (2022). Selenium [computer software]. Software Freedom Conservancy. https://www.selenium.dev/

Sen, R., Choobineh, J., Kumar, S. (2020). Determinants of software vulnerability disclosure timing. *Production and Operations Management, 29*(11), 2532–2552. https://onlinelibrary.wiley.com/doi/abs/10.1111/poms.13120

Shahzad, M., Shafiq, M. Z., & Liu, A. X. (2020). Large scale characterization of software vulnerability life cycles. *IEEE Transactions on Dependable and Secure Computing*, *17*(4), 730–744. https://doi.org/10.1109/TDSC.2019.2893950

Smeets, M. (2018). A matter of time: On the transitory nature of cyberweapons. *Journal of Strategic Studies*, *41*(1-2), 6–32. https://doi.org/10.1080/01402390.2017.1288107

U.S. Cyber Command. (2018). *Achieve and maintain cyberspace superiority: Command vision for U.S. Cyber Command*. https://www.cybercom.mil/Portals/56/Documents/ USCYBERCOM Vision April 2018.pdf?ver=2018-06-14-152556-010

U.S. Joint Chiefs of Staff. (2018). *Cyberspace Operations* (JP 3-12). https://www.jcs.mil/ Portals/36/Documents/Doctrine/pubs/jp3_12.pdf?ver=2018-07-16-134954-150

Warren, T. (2021, January 6). *Windows 7 is still running on at least 100 million PCs*. The Verge. https://www.theverge.com/2021/1/6/22217052/ microsoft-windows-7-109-million-pcs-usage-stats-analytics

You, I., & Yim, K. (2010). Malware obfuscation techniques: A brief survey. *2010 IEEE International Conference on Broadband, Wireless Computing, Communication and Applications, 297–300.* http://dx.doi.org/10.1109/BWCCA.2010.85

Zach. (2021, March 2). *An introduction to the exponential distribution*. Statology. https://www.statology.org/exponential-distribution/

Zyte. (2022). Scrapy [computer software]. Zyte. https://scrapy.org/

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California