"Watching over the shoulder of a professional": Why Hackers Make Mistakes and How They Fix Them

Irina Ford, Ananta Soneji, Faris Bugra Kokulu, Jayakrishna Vadayath, Zion Leonahenahe Basque, Gaurav Vipat, Adam Doupé, Ruoyu Wang, Gail-Joon Ahn, Tiffany Bao, Yan Shoshitaishvili
Arizona State University
{*iford3, asoneji, fkokulu, jkrshnmenon, zbasque, gvipat, doupe, fishw, gahn, tbao, yans*}*@asu.edu*

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00202

***Abstract*—The complex and diverse nature of software systems necessitates a careful manual approach to unveil vulnerabilities, involving deep analysis, creative problem-solving, and specialized expertise. Like all complex tasks, it's susceptible to mistakes stemming from cognitive limitations and behavioral factors that hinder optimal performance. Although there are significant research efforts focused on vulnerability discovery, little attention has been given to comprehending mistakes within the process. Understanding these mistakes could pave the way for better-designed education programs and automated tools, aiming to mitigate and prevent potential mistakes and enhance the efficiency of vulnerability research.**

In this paper, we leverage social media, specifically YouTube, to examine mistakes made by security content creators exploiting vulnerabilities in CTF-style challenges. Analyzing 30 screencasts from 11 hackers, we identified 124 distinct issues and investigated their types, underlying causes, and time investments. Additionally, we delved into the cognitive and behavioral aspects associated with these issues.

1. Introduction

The software that runs so many aspects of modern life—from our calendars to our pacemakers—is, unfortunately, rife with security vulnerabilities. Addressing these vulnerabilities before attackers find and exploit them is of utmost importance. In practice, white-hat security researchers find it necessary to exploit vulnerabilities for developers to take reports seriously [5, 43]. As Felix "FX" Lindner aptly put it, "It's hard to argue with a root shell" [23]. Thus, the white-hat security process involves not just vulnerability discovery but also vulnerability *exploitation*.

Despite advancements in automatic techniques, the exploitation of vulnerabilities is still heavily reliant on manual effort. This effort is non-trivial and highly technical: as discussed by the hackers in Votipka et al.'s study, exploit generation "can be the most time-consuming part of the [hacking] process" [43]. Anecdotally, this sentiment is well-supported in the community: hacking (the term that we use in this paper to refer to the two-step process of vulnerability discovery and exploitation) takes enormous amounts of time, further straining already overworked security researchers.

Interestingly, little research has explored *why* hacking is time-consuming, hindering progress in improving this situation. We, as long-time practitioners in both security education and security research, intuit that a large contributor to the time cost of hacking is the mistakes that hackers make: the process of hacking is extremely *error-prone*, and every error might lead a hacker down time-consuming, false paths. It follows that understanding these errors is imperative for enhancing the overall efficiency of hacking.

In this paper, we explore this intuition by carrying out the first study of mistakes made by hackers while hacking to understand *what* they are, *why* they happen, and *how* they impact the process. We leverage preexisting self-recorded YouTube videos of security content creators exploiting vulnerabilities in binary code. Our study focuses explicitly on mistakes made by 11 hacker subjects performing hacking activities in 30 screencasts. We analyzed each activity to identify mistakes, extracted behaviors associated with these mistakes, evaluated their impact, and studied how hackers commit, notice, and resolve them. Our study reveals that *all* subjects made at least one mistake, and the time cost associated with these mistakes consumes on average 40% of the entire process, validating our intuition and underscoring the need to understand (and develop solutions to prevent) these mistakes.

More broadly, we aim to answer the following research questions through our study and resulting analyses:

RQ1: What types of mistakes do hackers make in the hacking process (Section 4)?
RQ2: In what ways do these mistakes impact hackers, in the context of the entire hacking process (Section 5)?
RQ3: What are the non-technical factors that cause mistakes, and what techniques do hackers use to mitigate and avoid mistakes (Section 6)?

DOI 10.1109/SP54263.2024.00182

RQ4: How do hackers strategically handle mistakes (Section 7)?

By exploring the above questions, we made several novel observations related to mistakes in hacking. First, our study identified four different broad types of mistakes and found that the sub-category of "memory operation-related mistakes in exploit generation" was the most prevalent and time-consuming, accounting for a full 24% of total hacking time.

We also studied the causes of mistakes, with the most prominent (Memory Lapses, Unawareness of Strategy, and Lack of Attention) resulting in 23% of the total mistakes in our dataset. Once mistakes are made, their detection by hackers can be significantly delayed, especially early in exploit development, and hackers spend 12% of their time operating in the presence of undetected mistakes. We found that hackers spend a plurality of their time debugging mistakes (17% of total hacking time) and implementing fixes for mistakes (11% of total time). Our analysis of hackers' interactions with their audience indicates that collaboration can serve as a powerful tool for improving the efficiency of exploitation. Additionally, integrating advanced exploitation frameworks, such as pwntools can proactively address unforeseen issues and boost efficiency in exploitation.

This analysis of hacking mistakes suggests areas for improvement in educational and technical approaches to hacking. For example, the amount of time lost to mistakes in memory calculations in our study mirrors findings about student difficulties in memory management in programming [6, 22], and suggests that additional education effort in this area may benefit both software development and hacking. Furthermore, the hackers in our study employed a wide variety of disparate techniques in recovering from mistakes, hinting at promising impact from a future common "debugging curriculum." Finally, tooling and technique efforts invested in helping hackers verify and quickly remediate hacking mistakes can save significant time and effort.

Contributions. The contributions of this work are as follows:

- We study the types and causes of mistakes that humans commit during hacking, from vulnerability discovery to exploit development. To the best of our knowledge, this is the first study that investigates mistakes throughout the end-to-end hacking process.
- We develop a model of mistake identification and resolution and quantify the impact of every stage of the making and fixing of mistakes. This allows us to reason about improvements in the hacking process.
- We derive novel insights into hackers' mistakes and propose ways to enhance hacking education and future research.

To empower research in this space, we will release our annotated dataset for use by future researchers.

2. Background and Related Work

Human error research. Human error research has roots in psychology, cognitive science, and human factors engineering. It emerged from observations within complex systems such as aviation and industry, seeking to understand why errors happen and how to prevent them [18, 35, 45]. This field draws from various disciplines and intersects closely with decision-making models such as Decision Ladder Template (DLT) [32, 33] and Naturalistic Decision Making (NDM) [24]. In particular, DLT elucidates the decision-making and action-taking processes of individuals in dynamic and high-pressure situations. The model centers on the progression of individuals or teams through a series of information-processing stages when confronted with a problem. It also considers behaviors that may arise in varying circumstances, such as when experts encounter unfamiliar situations or when novices undertake diverse tasks. In contrast, NDM describes specific strategies that experts use to make decisions in familiar situations [29]. DLT has been employed across different domains to analyze human errors in industrial tasks, aviation, and healthcare [8, 15, 19, 21].

Our work, although distinct in scope, aspires to contribute to the broader landscape of human error research by offering unique insights into the realm of mistakes occurring in vulnerability research tasks. To that end, we adopt a modified version of the DLT model as a guiding framework, leveraging its insights to evaluate hacker mistakes. This utilization reflects the adaptability and applicability of DLT as a tool for understanding and addressing errors in complex and dynamic scenarios.

Mistakes in computer science. In recent years, numerous researchers have studied mistakes occurring in software development and security with a focus on practical solutions for identifying and addressing these mistakes. These studies employ various research techniques such as qualitative analysis, experimentation, and systematic literature review, to gather data and draw insights from real-world situations. Several studies aim to inform educational approaches and curricula, providing insights into common errors made by novice programmers, to improve teaching methods and learning outcomes [14, 22, 37]. Others are concerned with identifying mistakes leading to software vulnerabilities [16, 36, 41].

Human error in cybersecurity has predominantly been linked to security breaches [12, 31]. However, it is crucial to recognize that the impact of human error extends beyond breaches, permeating various facets of cybersecurity. Our work delves into the intricate domain of vulnerability research, shedding light on the technical and human aspects of errors in this complex and time-consuming field. Through this work, we seek to enrich the understanding of hacking mistakes and provide valuable perspectives that can augment the collective knowledge base, fostering advancements in the field and enhancing overall hacking efficacy.

Cognitive processes in vulnerability research. Cybersecurity research has increasingly focused on comprehending how the human brain processes information and makes decisions during vulnerability research tasks. Bryant [9] conducted multi-study research which included a case study, semi-structured interviews, and an observational study of reverse engineers performing specific tasks. His work resulted

in a seven-step model of sense-making in reverse engineering. Votipka et al. [42] performed an observational interview study of 16 professional reverse engineers and found that reverse engineering involves three distinct phases: overview, sub-component scanning, and focused experimentation, and provided insights about techniques used during each phase. Burk [10] attempted to quantify the understanding of binary programs in a large-scale experiment where participants aimed to achieve "perfect decompilation" and showed that it is within reach of human engineers. Mantovani [27] quantitatively measured the behavior of reverse engineering experts and novices during the analysis of assembly code via the use of Restricted Focus Viewer [20]. Our study contributes to the ongoing exploration of cognitive processes within the vulnerability research domain. Offering a distinctive perspective, we delve into the analysis of mistakes occurring in complex security tasks, seeking insights into the associated cognitive and behavioral patterns.

YouTube for knowledge communication. The YouTube video platform has been a rich source of freely accessible and valuable artifacts for research. Snelson examined the diverse ways in which YouTube was utilized in various academic fields through an extensive literature review [38], where it analyzed existing research to understand how YouTube was integrated into teaching, learning, and research across different disciplines. Several publications studied the impact of YouTube on education, its role in enhancing learning, the engagement and interactivity facilitated by this platform, and the challenges that come with its use [11, 44]. MacLeod et al. [26] performed a qualitative study on how software developers leverage YouTube to document and share their programming knowledge. The study focuses on how developers create video content, tutorials, and demonstrations to convey coding concepts, problem-solving techniques, and programming tips to a wider audience. Most relevant to our research is the article "YouTube Security Scene" by security content creator LiveOverflow [25]. The author shares their experience of "watching over the shoulder of a professional" and how the exposure to the problem-solving process of a YouTube streamer was instrumental in overcoming personal learning obstacles:

> *To see how geohot was using the terminal, writing exploit scripts, and navigating IDA Pro was incredibly insightful. But more importantly, it also exposed the fails and mistakes followed by the process of troubleshooting and fixing the bugs. And this pushed me through the wall I was hitting in my own education.*

In our study, we use YouTube to create a distinctive dataset on binary exploitation challenges. Most importantly, by diverging from conventional methods (e.g., surveys, think-aloud, and interviews), our novel approach provides conditions that resemble real-world experiences and reduce biases such as demand characteristics and the Hawthorne effect present in conventional methods [7].

3. Method

Our primary goal was to investigate the mistakes made by hackers during the evaluation and exploitation of vulnerabilities. To achieve this, we utilized a qualitative approach involving thorough analysis of 30 YouTube video recordings, offering valuable visual insights into the hacking process. Additionally, we analyzed the accompanying audio narratives from collected screencasts. Such a study design allowed us to conduct a rigorous examination of both technical aspects and contextual insights of hacking mistakes. This study received an exempt determination from Arizona State University's Institutional Review Board (IRB) since the data collection and analysis were conducted on publicly available YouTube data, thus further obviating the need to contact the hackers.

In this section, we describe the data collection process, the mistake framework that we used to conduct our work, the data analysis methods, and the limitations of our study.

3.1. Data Collection

TABLE 1: Demographics of hackers involved in the study (as of September 2023).

HID	Source	Chall solved	Joined YT	Last video	N subscr	N vid
H1	Keyword search	5	2018	2019	69	8
H2	Keyword search	5	2011	2023	4.04K	447
H3	securitycreators*	1	2007	2022	8.86K	41
H4	securitycreators	2	2013	2023	42.2K	149
H5	Keyword search	4	-	-	-	-
H6	securitycreators	8	2013	2020	2.89K	82
H7	securitycreators	1**	2013	2023	2.03K	41
H8	Keyword search	1	-	-	-	-
H9	securitycreators	1	2017	2020	15.3K	70
H10	securitycreators	1	2015	2023	835K	413
H11	Keyword search	1	2006	2021	9	48

* https://securitycreators.video
** Hacker solved the same challenge twice one year apart

YouTube videos selection. Our goal was to curate a diverse yet representative sample, offering insights into difficulties encountered by hackers in vulnerability recognition and exploitation. For our research, we targeted YouTubers recording themselves solving Capture-The-Flag (CTF) style challenges on war-gaming platforms such as PicoCTF [2], pwnable.kr [3], OverTheWire [1], and others. Our inclusion criteria encompassed videos a) featuring binary exploitation challenges, b) solved without previously seeing the challenge (*blind-solving*), and c) where hackers made at least one mistake while solving the challenge.

We started our search for YouTube videos with the identification of certain keywords. Our keywords list contained terms and phrases like *live hacking*, *hacking livestream*, *binary exploitation*, *pwnable*, and *PicoCTF*. With this list, a preliminary search on YouTube allowed us to identify most of the hackers for our study. In addition, we conducted a quick scan through the Security Creators website [4], where hackers' short descriptions and media links were posted. Some streamers advertised channels with similar content in their videos, which we also used in our search. Following

this process we selected 11 hackers for our study. Table 1 provides essential hacker demographics as of September 2023.

In total, we were able to identify more than 90 videos on the selected hackers' YouTube channels pertinent to our research. At this stage, we performed a more detailed scan through videos to apply our inclusion criteria. Notably, to satisfy the blind-solving (hackers should not have seen or solved the challenge before) inclusion criterion, we relied on explicit statements made by the streamers or observed their behavior.

Of the 11 hackers we selected for the study, seven explicitly noted that they were solving the challenge without previous knowledge. The remaining four hackers implied they were blind-solving. One of the hackers stated they were skipping a known challenge, implying they only solved unknown challenges in their video. Another hacker was participating in a competition with rules for only blind-solving challenges. The last two hackers had no implicit indicators; however, they are widely known for posting videos of blind-solving challenges.

Following the selection process, a final sample of 30 YouTube video streams created by 11 hackers with total footage of 61 hours was chosen for in-depth analysis.

Video descriptions. Screencasts were conducted by hackers spanning the period from 2014 to 2020. Most videos were created in Full HD (1080p) with good audio quality. One video had a resolution of 480p, and two videos exhibited high background noise, thus providing a low level of clarity and detail and complicating analysis. 30 screencasts contained 31 challenges (two challenges were solved during the same stream) from CTF competitions or war-gaming platforms. Only 28 of them were unique as two challenges were solved by two different people and one person solved the same challenge one year apart. More than half of the streams ($n = 18$) contained other content not relevant to our study. The time of solving a challenge varied from 8 minutes to 3 hours 7 minutes with the total time of analyzed video material amounting to 35.6 hours.

Challenges included x86 and x86-64 vulnerable Linux binaries with varying degrees of protection. Source code and libc files were provided in 13 and 6 challenges correspondingly. To solve the challenge streamers had to understand the binary, identify vulnerability, and exploit it to gain control over the program: spawn a shell, or retrieve a specific flag. The difficulty of a particular challenge was hard to assess since streams were created over 7 years, however, they can be categorized as beginner-intermediate level. Challenges covered a spectrum of memory corruption vulnerabilities, such as arbitrary read and write, stack buffer overflow, format string, heap unlink and use-after-free, stack frame corruption, and stack-use-after-return. Detailed characteristics of selected videos and challenges are presented in Table 2.

During the solving process, streamers used either a local or remote (server) environment. While working in a local environment all streamers used a standard set of applications to solve challenges, which included a web browser, terminal, disassembler/decompiler suite (IDA, Hopper, Binary Ninja, Radare), debugger, and text editor (IDE). Depending on their needs they could use utility applications such as calculator, paint, and hex editor. In several instances, hackers had to perform particular tasks remotely and utilize the software available on the server, usually GDB without extensions and Vim editor. Most of the time hackers had to write a Python script to exploit a vulnerability unless they could use a bash one-liner or just supply a short numeric input while interacting with the challenge (e.g., format string vulnerability).

Ethical considerations: All YouTube videos are part of the public domain and can be found online. While we do not disclose hacker names, the identities can still be inferred due to the public nature of used artifacts.

3.2. Mistakes Analysis

We conducted both technical and qualitative analyses on the screencasts to address our research questions.

Technical analysis. We used a multi-phase process to thoroughly analyze screencasts for the contained mistakes. Initially, the primary researcher identified mistakes, determined their timeframes using the mistake anatomy framework (Section 3.3), and defined root causes. Subsequent verification and validation processes were implemented to ensure accuracy and reliability. The verification phase focused on confirming the existence and nature of identified mistakes, while the validation phase probed the underlying root causes.

During the verification phase, we selected one mistake in each video using simple random sampling and delineated a 10-minute video snippet containing this mistake. In 15 out of 31 cases, these snippets featured two or three mistakes. Two expert researchers with top-level CTF experience independently verified 48 issues identified by the primary researcher. This process uncovered three new mistakes and combined two, totaling 50 (40%) of the 124 issues. The agreement percentage among the three researchers determined the reliability. Disagreements were addressed at reconciliation meetings until more than 90% of agreement was reached.

This rigorous verification helped in consensus-building and set the stage for the validation phase, where one expert researcher performed validation of the mistakes timeframes and root causes by reviewing all videos.

We classified identified mistakes based on the root cause and applied different visualization techniques to uncover inherent trends and patterns and answer our research questions (RQ1-2).

Qualitative analysis. To better understand cognitive aspects of hacker mistakes and streaming behaviors, we analyzed narratives from the videos using an iterative open coding process [39]. Initially, we randomly selected five challenges from five different hackers and three researchers individually examined the video transcripts to create individual codebooks. Then researchers met to discuss their codebooks and designed a unified codebook with an attempt to group open

TABLE 2: Characteristics of all hacking videos and the challenges involved in these videos.

HID	CID[1]	Wargame/CTF platform	Year streamed	Res[2]	Views count[3]	Challenge duration[4]	Arch	Source code / protections[5]	Vulnerability
H1	C1	pwnable.tw	2019	1080	332	1:36:00	x86	C, NX	arbitrary read and write
	C2		2019	1080	193	1:28:00	x86-64	NX, stripped	arbitrary write
	C3		2019	1080	185	0:56:00	x86	C, NX, stripped, libc	heap use-after-free
	C4		2019	1080	112	0:57:00	x86	R, NX, libc	stack buffer overflow
	C5		2019	1080	137	1:21:00	x86	C, NX, libc	stack-use-after-return
H2	C6	pwnable.kr	2018	1080	1,119	0:55:00	x86	C, NX	stack frame corruption
	C7		2017	1080	1,162	0:43:00	x86-64	S, C, NX	heap use-after-free
	C8		2018	1080	1,165	1:21:00 (2:37:00)	x86	S, NX	heap unlink
	C9		2018	1080	580	1:17:00 (1:26:00)	x86	C, NX, libc	stack buffer overflow with ret2libc
	C10		2018	1080	554	2:39:00 (7:04:00)	x86	C, NX	stack buffer oferflow, canary bypass
H3	C11	custom	2020	1080	782	0:25:00	x86-64	NX	stack buffer overflow
H4	C12	PicoCTF	2019	1080	6,111	0:28:00	x86-64	S, NX	stack buffer overflow
	C13		2020	1080	3,817	2:42:00	x86	S, C	heap use-after-free
H5	C14	OTW	2014	480	700	0:12:00	x86	S, ?	out of bounds memory access
	C15	pwnable.kr	2014	1080	917	1:00:00	x86	C, NX, libc	stack buffer overflow with ret2libc
	C16	OTW	2014	720	93,550	0:24:00	x86	NX	stack buffer overflow
	C17		2014	720	93,550	1:05:00	x86	NX	format string with GOT overwrite
H6	C18	PicoCTF	2019	1080	317	0:16:00	x86	S	arbitrary write
	C19		2019	1080	162	0:14:00	x86	S, NX	stack buffer overflow
	C20		2019	1080	135	0:40:00	x86	S, NX, PIE	stack buffer overflow with ret2libc
	C21		2019	1080	179	0:25:00	x86	S, NX	format string
	C22		2019	1080	208	0:08:00	x86	S, NX	arbitrary write
	C23		2020	1080	225	2:13:00	x86	C, NX	format string with GOT overwrite
	C24		2020	1080	126	0:45:00	x86-64	S, C	arbitrary write
	C25		2020	1080	361	1:48:00	x86-64	S, C, NX, libc	heap use-after-free
H7	C26	RopEmporium	2018	720	1,076	1:13:00	x86-64	NX	arbitrary write
	C27		2019	1080	1,010	1:53:00 (1:58:00)	x86-64	NX	arbitrary write
H8	C28	custom	2018	720	1,259	3:07:00	x86-64	R, C, NX, PIE	out of bounds memory access
H9	C29	PlaidCTF	2017	1080	4,020	1:05:00	x86-64	S, R, NX, PIE, libc	out of bounds memory access
H10	C30	OTW	2015	1080	28,861	0:54:00	x86	S, ?	out of bounds memory access
H11	C31	Nightmare	2020	1080	55	1:19:00	x86	R, NX, PIE	stack buffer overflow

[1] C9/C15 and C14/C30 are the same challenges solved by two different hackers, C26/C27 is the same challenge solved by one hacker, all challenges correspond to individual streams except for C16 and C17 that were solved during one stream.
[2] Video resolution (pixels). C9/C10 challenges have high resolution but poor audio quality.
[3] Time-sensitive information (reflects the number of views on the date of extraction).
[4] Time in brackets indicates challenge duration with added pause time if a hacker paused video during stream.
[5] C - Canary, NX - Non-Executable stack, PIE - Position Independent Executable, R - full RELRO, S - source code present, libc - libc file provided, stripped - symbol information removed from binary, ? - information not available.

codes, which they subsequently employed to independently code an additional 6 challenges from the remaining hackers. Following that, researchers extensively discussed the new codes and generated themes, resolving any disagreements and refining the codebook accordingly, which helped increase the validity of themes. Subsequently, to assess the refined codebook's applicability, researchers coded five new challenges and revisited the initial five, aligning their coding with the established codebook. Leveraging an iterative process, researchers then refined the codebook by removing and merging themes until a consensus on the final and overarching themes was achieved. The final codebook, derived from the iterative coding process of 16 challenges, incorporated these finalized themes. The primary researcher utilized this codebook to code all remaining challenges.

We employed an inductive approach to qualitatively analyze hacker streaming behaviors, with a specific focus on mistake timeframes [40]. This approach was deemed appropriate for our study due to the absence of prior exploration in this domain. Furthermore, since multiple researchers reviewed the codes and themes and critically conducted detailed discussions at all stages of the analysis, we do not report inter-rater reliability. According to a meta-analysis of CSCW and HCI publications from 2016 to 2018, only 10% of papers using iterative open coding with multiple coders measured inter-rater reliability [28].

Due to the qualitative nature of our results, we present findings in Sections 6 and 7 qualitatively, sparingly offering counts of themes to emphasize pattern prevalence. It is crucial not to infer the prevalence of themes beyond the sample.

3.3. Mistake Anatomy

To comprehensively describe and characterize mistakes made by hackers, we adapted Rasmussen's DLT model of information processing steps of the industrial process plant operator to handle errors [34]. The original DLT model primarily focuses on describing the sequence of information processing and decision-making activities that occur during human-machine interactions. It outlines stages such as detection, observation, interpretation, evaluation, and execution, providing a framework for understanding how errors can propagate through these stages. Based on this model we construct *mistake anatomy* (Figure 1), a framework for detailed documentation and analysis of key hackers' activities while they try to resolve their issues. The framework serves as a structured guide for dissecting a mistake starting from the time when it was made by a hacker, through the earliest point where the effects of the mistake can be noticed and

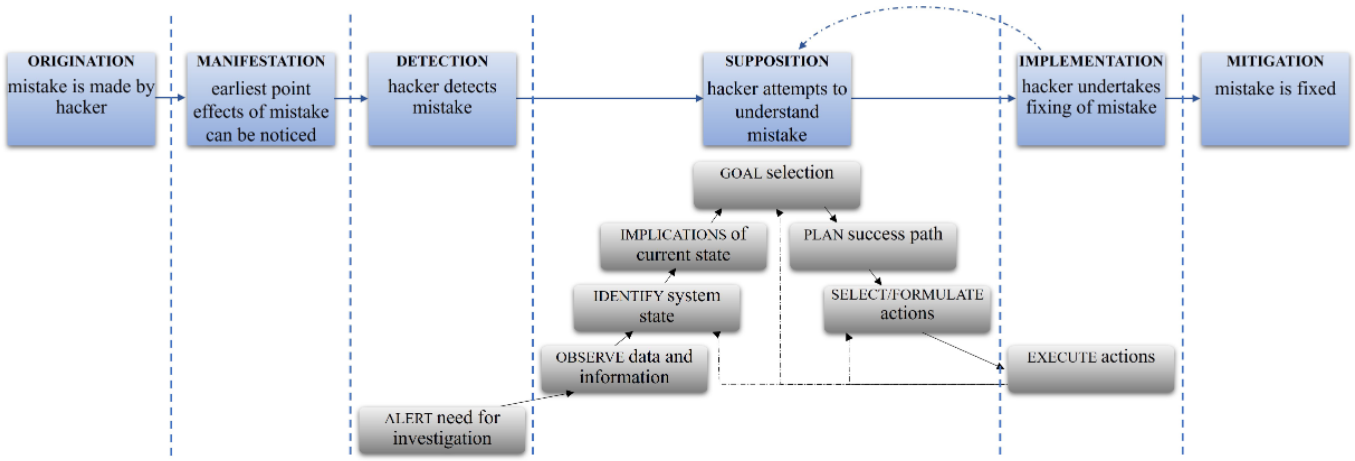

Figure 1: Mistake anatomy fit to the Decision Ladder Template (Rasmussen, 1974).

detected by a hacker to the attempts to understand the issue, undertaking to correct it, and finally, to the time the mistake is fixed. The key terms for the framework were crafted through a detailed examination of several mistakes:

Origination: Hacker commits a mistake. To identify this phase we track back the changes made by a hacker (e.g., in code) from the observed mitigation point.

Manifestation: The earliest point where the mistakes' effects become evident, displaying errors, incorrect output, or the absence of expected outcomes on the hacker's screen.

Detection: Hacker realizes an error.

Supposition: Hacker attempts to understand the issue cause.

Implementation: Hacker tries to implement a solution based on their understanding of the problem.

Mitigation: Hacker successfully executes exploit.

DLT is a general theoretical model for understanding the flow of information and decisions, thus it does not emphasize the root causes or origins of mistakes, the specific nature of the error manifestation, or the mitigation of mistakes. In our analysis, we focused on identifying root causes and describing temporal relationships between different phases in a mistake life cycle as well as determining the impacts of mistakes. To enhance the framework, we integrated aspects related to the origin, manifestation, and mitigation of mistakes. The correspondence between different phases of our mistake anatomy framework and DLT is shown in Figure 1. Detection and implementation phases relate to alert and execute counterparts in Rasmussen's model, but the supposition phase correlates with a series of activities from observing information through selecting actions, which we considered as activities related to debugging a problem. We further hypothesized that supposition and implementation phases form an iterative loop that could be correlated with the feedback loops of DLT and that the supposition phase can be bypassed in the process of resolving a mistake similar to bypassing certain stages in the DLT model.

3.4. Limitations

While our study provides valuable insights into the mistakes made by hackers during vulnerability discovery and exploitation, we acknowledge certain study limitations.

Data collection time frame. Despite our attempts to perform a comprehensive search and collect all videos meeting our inclusion criteria we still might have missed some videos. The data collection phase stopped in 2021, however, we continued monitoring YouTube for the appearance of study-eligible videos until July 2023. Although many new hacking streams emerged from 2021 to 2023, we found that they were either tutorials or walkthroughs, or were related to other security fields such as web or network security, and thus did not match our inclusion criteria. We also stopped collecting and analyzing more videos when theme saturation was reached.

Self-reporting. One criterion for the selection of videos was solving a challenge without prior knowledge or information about it. Usually, hackers assert blindness and we have to trust them. Blindness is constrained by various factors: knowing vulnerability beforehand or reading write-ups during the stream, pausing the video to do research or rest, collaborating with viewers, and asking for and getting help from the audience. In one case a hacker worked on a challenge that he previously resolved but was dissatisfied with the subpar solution.

Researcher subjectivity. Identifying timeframes of different activities during resolving a mistake carries a subjectivity component, especially when trying to distinguish between the debugging and implementation phases, which are often tightly intertwined. During the analysis of videos, we often observe the effect of compound mistakes (several mistakes committed while solving a challenge), or overlapping mistakes (one mistake committed during resolving another issue). These effects complicate both technical and qualitative analyses by making it challenging to clearly distinguish between timeframes / qualitative codes relating to particular mistakes.

Observational study. Due to the observational nature of our research, we can only partially perceive our subjects' thoughts and emotions through their expressions.

Threats to validity. The use of CTF-style challenges as a proxy for real-world security tasks may impact the generalizability of our findings, as challenges are characterized as beginner/intermediate-level CTF tasks and are likely less complex than real-world code bases. However, due to the complexity of the real-world code, our analyses would not be scalable given our study's granularity. While we attempted to include a diverse set of hackers, it is essential to acknowledge that not all may be considered industry-grade hackers.

Moreover, we identify certain limitations stemming from our hackers' streaming behaviors impacting study validity: 1) the necessity to provide a solution within the time constraints of a stream may introduce a bias in the types and amounts of mistakes made. Hackers might prioritize speed over thoroughness, impacting their hacking, 2) the act of narrating while streaming could also potentially influence the types and amounts of mistakes made. The presence of an audience and the need for clear communication may lead hackers to articulate their thought processes more deliberately than they would in a non-streaming context, and lastly, 3) viewers' suggestions, comments, or interactions could potentially influence the overall hacking experience and hackers' decision-making processes.

4. Technical Analysis—What Mistakes?

Through analyzing 30 video streams, we recognized 124 issues that hackers had to address when solving challenges. Among these, we identified 117 as mistakes and organized them into four types depending on their root causes: *Programming/Miscellaneous*, *Implementation*, *Strategy*, and *Tooling*. We categorized the remaining issues as technical difficulties and combined them into a *Technical Issues* type. Table 3 in Appendix A presents the issue categories and their counts. For each issue, we were able to identify a root cause (RQ1) and determine a set of timeframes according to our mistake anatomy framework. We used these timeframes to estimate the duration of different phases of mistake lifetime and evaluate the costs and impacts of these issues (RQ2). Figure 2 presents the distributions of types of mistakes.

Next, we describe all mistake types and their temporal characteristics. We categorize each mistake type into subtypes for a more detailed understanding of issues within each category.

4.1. Implementation

We categorize mistake as *Implementation* if it pertains to memory operations, such as failures when reading from or writing to memory. These operations are critical in enabling attackers to disclose or overwrite memory values at controlled addresses during the exploitation of memory corruption vulnerabilities. We discovered 50 such mistakes

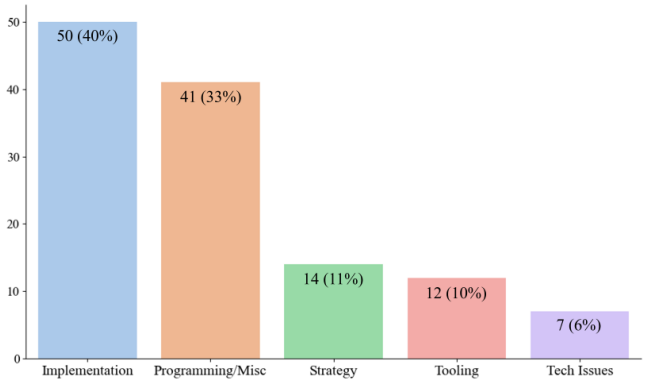

Figure 2: Distribution of mistake types.

and organized them into four classes: *leak*, *write what*, *write where*, and *write what/where* depending on their specific actions and objectives.

Primary causes of *leak* errors are incorrect format string specifiers, miscalculation of offsets from leaked library addresses, or incorrect calculation of addresses from leaked values. The median time to resolve such mistakes was 7 minutes. The most prevalent type, *write what* errors (72%), include problems with payload crafting, function argument requirements, use of the wrong heap object, prohibited overwrites, missing gadgets in ROP chains, and stack misalignment issues. The median resolution time for these errors was 7 minutes, with two outliers taking over 50 minutes. *Write where* errors (8%) comprise mainly mistakes in offset to overwrite a pointer, with a median resolution time of 7 minutes and cases ranging from 1 to 52 minutes. Lastly, four instances of *write what/where* mistakes averaged an 11-minute resolution time, encompassing cases of miscalculated offsets and wrongly constructed ROP chains. More details can be found in Appendix A.1.

4.2. Programming/Miscellaneous

One-third of all the identified issues belong to the *Programming/Miscellaneous* type. These mistakes include three sub-types: runtime errors, syntax errors, and incorrect logic and oversight. More than half of all programming mistakes (n = 19) are runtime errors. Hackers commonly use undefined variables and functions and wrong type conversions (sometimes resulting from typographic errors), apply improper functions to data structures, mishandle arguments, or misread Python help documentation, resulting in Python exceptions and pwntools exceptions. Figure 3 shows an example of a conversion error where a hacker forgot to pack an integer into binary form.

We noticed three syntax error cases. The first two concerned forgetting to include a header file and using an undeclared variable. In the third case, the hacker used the `printf` library function without a format string specifier, which caused unexpected compiler optimizations to replace

```
1  ...
2  alloc(8, p32(0xXXXXXXXX) + elf.got['puts'])
3  ...
```

Figure 3: A hacker forgot to use p32() to convert the address of the puts() (an integer) into its binary representation in little-endian format.

`printf` with `puts` in the binary. The hacker did not realize this error until his audience pointed at it in chat.

13 programming mistakes involved incorrect logic and oversight. They include an instance of an infinite loop due to forgetting to decrement a counter, exploit logic mistakes such as leaving old code in the exploit script, misplacing code lines, omitting necessary function calls, or passing incorrect arguments to function calls. Several errors emerged from misusing pwntools functionality of send and receive operations. Hackers spent up to 15 minutes to resolve each mistake with a median time of 1.5 minutes. Most issues took under 5 minutes to resolve.

The *Miscellaneous* type mostly includes user input errors. We identify three Linux command-line input mistakes, where hackers forgot to use the pipe symbol when feeding the input file to the binary, passed the wrong argument to the OneGadget tool [13], and neglected to keep `stdin` open, resulting in the shell exiting. Other mistakes included inconsistent manual input due to typographical errors and mentally miscalculating numbers. Except for an outlier (21 minutes), mistakes in the miscellaneous sub-type were resolved within 4 minutes with an average time of 2 minutes.

We often observed situations where minor programming issues or oversights occurred as side effects while streamers were solving other problems (e.g., implementation mistakes). Due to the difficulty of separately evaluating these side effects, we did not account for them in our analysis and only considered standalone issues.

4.3. Strategy

We consider a mistake as a *Strategy* mistake if any of the following conditions occurred: (1) a hacker injected a poorly crafted payload that did not execute as expected, resulting in a failure to gain control over the system; (2) a hacker applied an exploit that was incompatible with the vulnerability; or (3) a hacker failed to trigger the vulnerability in the application. We identified 14 such mistakes made by five hackers in seven challenges. All these mistakes were abandoned as hackers had to re-think their approach. Vulnerabilities, where strategy errors were involved, include stack buffer overflows with ret2libc attack, canary bypass, unsafe unlink, format string with GOT overwrite, arbitrary memory write, and out-of-bound memory access.

Most strategy flaws (n = 10) fall into the first subcategory, where hackers failed to create payload. Sometimes, hackers tried to achieve arbitrary code execution by leveraging "magic gadgets" (instruction sequences in libc that will spawn a shell when executing) but some attempts were unsuccessful due to unmet conditions, and hackers had to reconsider the design of their exploits. In another case, the constructed payload exceeded the allowed size limit for a buffer overwrite. Other examples include corrupting a pointer in GOT when returning to the `main` function, creating a payload with the pointer in a non-writable segment, constructing a payload that reads pointers as strings but not accounting for the terminating null byte, and overcomplicating the solution by implementing a complex ROP chain.

Incompatibility issues occurred when hackers encountered a challenging task and did not have a clear direction for exploitation. In such situations, one approach was to research the topic off-stream and attempt to implement ideas they found. For example, while analyzing a buffer overflow with a canary bypass video, we encountered two cases where a hacker implemented techniques that either required the canary to be constant or granted data disclosure (leak) but not memory modification (write) capability. In another situation, a hacker was planning a partial pointer overwrite but had to abandon the idea due to a double-word write operation constraint.

Finally, we observed one case of selecting a faulty trigger for an exploit when a hacker had to reject the original plan of using `printf` to trigger `malloc` and used `scanf` instead. Hackers spent 9 minutes on average but up to 23 minutes total on these types of mistakes.

4.4. Tooling

Any mistakes associated with the wrong usage of tools like disassemblers, decompilers, or debuggers fell into the *Tooling* category. We identified 12 such mistakes and organized them into four sub-types.

Five mistakes occurred due to issuing incorrect instructions or failing to pass proper input to the binary in a debugger or disassembler, causing tool-specific errors or unintended output. Examples of utilizing wrong commands in GDB include setting `follow-exec-mode` instead of `follow-fork-mode`, using qwords instead of bytes for indexing, and `return` instead of `finish`. One hacker misread help instructions in Radare and had to quit out of the disassembler to avoid data loss. We also observed one case of misinterpretation of a string value during the reverse engineering of an assembly code.

Four mistakes involved misconfigurations and missteps in process control while using GDB. One of them related to an attempt to attach GDB to the process running under the context of a different user instead of creating a local copy first, leading to permission-related issues. Figure 5 shows this issue as a commented code in line 3. Two mistakes were associated with GDB following the wrong process due to hackers forgetting to set the corresponding option. Finally, one of the hackers attempted to run the wrong program in a debugger instead of running in fork mode as a child process.

Debugging oversight occurred twice while hackers were using GDB. The first time, a hacker looked for heap ad-

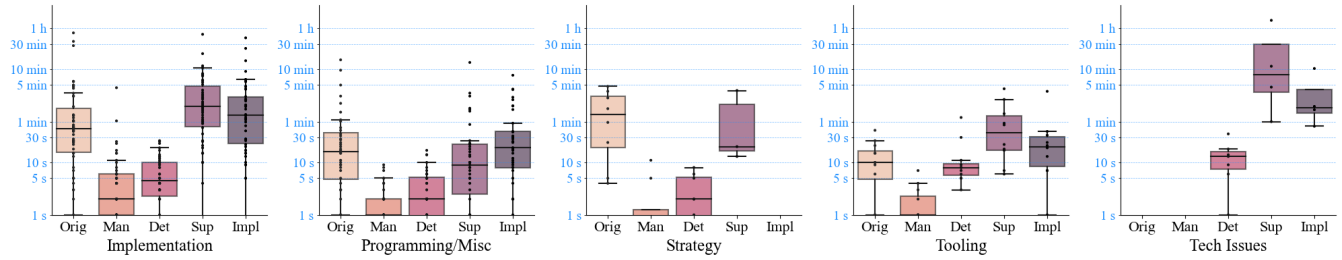

Figure 4: Distribution of phase duration for different types of issues. The x-axis labels denote the time difference between two points: Orig - origination/manifestation; Man - manifestation/detection; Det - detection/supposition; Sup - supposition/implementation; Impl - implementation/mitigation. The y-axis represents time in logarithmic scale.

```
1  import pwn
2  ...
3  # p = pwn.process('/problems/problem1/vuln')
4  p = pwn.process('./vuln')
5  ...
```

Figure 5: Originally did not use the local copy of the binary when trying to attach GDB.

dresses in memory before making an allocation; in the second case, a hacker attempted to read GOT entries in the read-write segment of code in a binary compiled with full RELRO. We noticed one issue related to setting a breakpoint in GDB on an instruction that was already executed, which prevented backtrace and continuation of analysis for 4.5 minutes. In general, *Tooling* mistakes were resolved quickly, with an average time of 2 minutes. The highest time loss (5 minutes) happened during solving a permission-related problem.

4.5. Technical Issues

Occasionally, hackers had to address technical issues that prevented further activities until resolved or abandoned. These issues cannot be characterized as mistakes. We identified seven such issues caused by complications with software, such as pwntools, the `pty` Python module, and the pwndbg GDB extension; missing dependencies for running 32-bit executables; and random program behaviors. Four issues were addressed by either implementing an alternative approach, installing missing dependencies, or testing varied assumptions. We could identify only detection, supposition, and implementation timeframes for those instances. The remaining three issues were abandoned. In two instances (pwntools did not work on the server, missing libraries), the streamer was able to, quickly, in under two minutes, find an alternative solution and continue. In other cases, hackers successfully installed required dependencies and modified code to control the process, spending 7 minutes on average. Complications with pwndbg caused one hacker to lose 20 minutes while trying to research the problem and patch the source code before he abandoned the issue because of time considerations. Another hacker spent more than an hour and a half "fighting with the weird implementation of pseudo pty inside Python libraries." As he confessed later:

> *I didn't expect master pipe to be interpreting any terminal sequences. Why would it? ... but it seems that library did it, and it changed what we sent to the process to gibberish... and we spotted that only after doing the tee into a file, ... and then we had to rewrite it to use low-level pseudo terminal implementation... I learned... not to trust the pty library, because it does weird stuff* (H4)

5. Cost-Impact Analysis of Mistakes

To better understand how the time costs of mistakes are distributed across their lifetime, we calculated the duration of phases occurring between each pair of timeframes in our mistake anatomy framework. We analyzed the distributions of these phases with respect to the issue type and how significant the cost of each phase was during the process of resolving issues. We discuss the case of *mistake abandonment* (i.e., when hackers give up on resolving an issue) in Appendix A.2. Figure 4 shows the distribution of phase durations for different categories of issues. Using log time on a vertical scale allows for uniform plotting across all graphs, highlighting that the most significant costs were accrued during three specific phases: origination, debugging, and fixing. Below, we describe each phase separately and compare them across various types.

Origination/manifestation. This phase represents the time it takes for a mistake to manifest or become visible on the screen. We were able to notice the effects of mistakes after 2 minutes on average, however, the median time was only 24 seconds, indicating a positively skewed distribution with relatively few high values. Maximum duration times varied for different types, reaching 15 and 49 minutes (98% and 95% of the total duration time, respectively) for *Programming* and *Implementation* types. Elevated values could be attributed to two scenarios: a) mistakes originating early in the process of writing the exploit and remaining unnoticed until testing; b) errors involving multiple compounded mistakes, some of which surfaced and required resolution before the original mistake could manifest. Frequently, both scenarios coexisted.

Manifestation/detection. This phase indicates the time required for a hacker to detect a mistake. This occurrence typically took place shortly after a mistake appeared on the screen, with both the average and median times estimated in the order of several seconds. In one instance, this phase lasted 4.5 minutes (29% of the total time). The hacker constructed a payload for the write using the GOT address of the system instead of PLT but was not immediately able to recognize the issue as the binary displayed the expected behavior (segmentation fault) during interim testing. We could not estimate this phase in cases of the wrong approach where hackers realized their mistake at the conceptual level.

Detection/supposition. This time interval represents how soon it takes for a hacker to start making assumptions after detecting a mistake. Similar to detection, this phase was very short and took only several seconds on average, as streamers usually began investigating the reasons shortly after they had detected the mistake.

Supposition/implementation. This is a debugging phase during which hackers attempt to understand the problem and test their hypotheses. It lasted 3.3 min on average, with a median time of 1 minute, indicating a positively skewed distribution with a few outliers, including a maximum value of 1 hour 25 minutes spent on resolving an issue with the Python `pty` library. The phase lasted the longest (4 minutes on average) for *Implementation* mistakes, followed by *Strategy* (1.5 minutes). For *Tooling* and *Programming* types, it took approximately 1 minute. *Technical Issues* are characterized by a median time of 1 minute but show a high average (25 minutes) due to an outlier. There were only three instances for strategy and four for technical problems where we were able to estimate this phase due to a high number of mistakes abandoned at this stage.

Implementation/mitigation. During this phase, hackers attempted fixing the mistake. This phase lasted 2.3 minutes on average, with a maximum time of 40 minutes where a hacker attempted to implement a solution "on the fly" neglecting to understand the issue first. Similar to the debugging phase, the duration was the longest for *Implementation* mistakes with an average time of 3.5 minutes. The distinction between supposition and implementation phases is blurred as hackers frequently endeavored to implement ideas that did not succeed, leading them to reassess the problem and engage in an iterative loop of understanding and experimentation. Figures 6 and 7 show distributions of time required to resolve different mistake types and subtypes correspondingly.

Figure 8 demonstrates the percentage of time lost on mistakes and technical problems in all challenges. Time costs are sorted in increasing order with the type of vulnerability shown on the left and the hacker ID on the right of the bars. Time losses vary in the range from 3 to 75% with the average value of 40%. In 30% of all cases (n = 9), the time lost accounts for more than 50% of the total challenge duration.

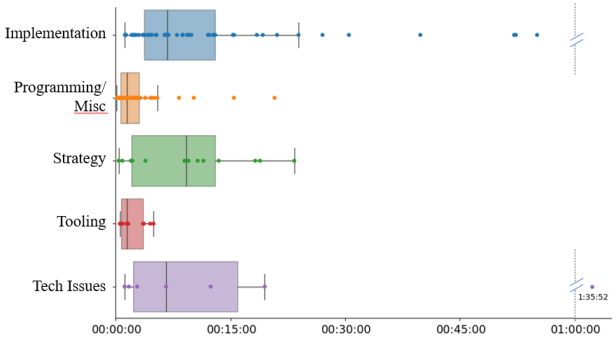

Figure 6: Distributions of resolving times for different types of issues.

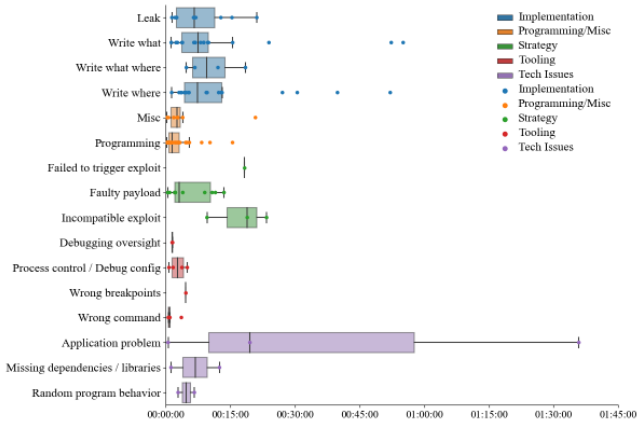

Figure 7: Distributions of resolving times for issue subtypes.

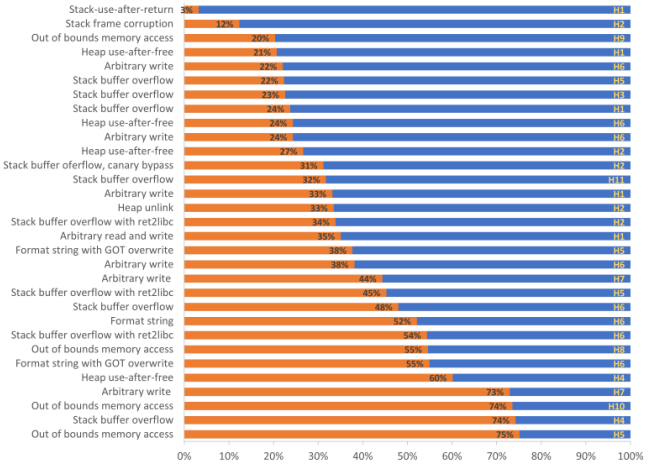

Figure 8: Percentage of time lost on rectifying issues. Time spent on resolving mistakes and technical issues (orange) is shown as a percentage of the total time spent on the challenge (orange + blue).

6. Behind Mistakes—Why Making Mistakes?

According to cognitive psychologists Rasmussen [32] and Reason [35], human errors result from normal cognitive processes. While technical analysis helped uncover hacker mistakes and their root causes, in this section we delve into the non-technical reasons behind those mistakes along with debugging and avoiding mistakes approaches.

6.1. Making Mistakes

We identified various non-technical factors contributing to mistakes through qualitative analysis of video transcripts. Six hackers (H1-4, H6, H7) made mistakes due to **unintentional overlook or forgetting critical elements**. These mistakes included missing gadgets while implementing a ROP chain, forgetting to assemble shellcode, failing to meet argument requirements, and neglecting to define or call variables or functions. Such occurrences are typical for *Programming* and *Implementation* types of errors. Often, hackers admitted these lapses:

> *I forgot to do buf2 plus equals, that's gonna happen a lot with buf... this is a horrible way of ROP building, by the way, and it leads to mistakes, like that one right there* (H7)

Unawareness of approach was usually associated with the *Strategy* type of mistakes. Four hackers (H2, H5, H7, H10) struggled with identifying the correct strategy to solve a challenge. For example, before attempting to leak the canary value, H2 confessed:

> *I've no idea if this is going to lead anywhere.*

We observed similar behavior during the implementation of memory write operations if a hacker (H5, H10) was misunderstanding pointer operations or payload constraints.

We also noticed three cases of **over-complicating** the solution (H6, H7, H10). For example, H7 admitted:

> *I made this way more complicated than it has to be, way more complicated.*

Inconsistent hand included "fat-fingering" mistakes (essentially, typographical errors) (H6), copy/paste (H6, H8) or user input errors (H9). Some *Tooling*, *Programming*, and *Implementation* mistakes occurred due to the **lack of focus or attention**, for example, during reading documentation files (H4, H11), interpreting assembly code (H8), or writing an exploit (H1, H5, H7). In some instances, hackers (H10, H11) did not know **how to use tools** properly and confused themselves using the wrong commands. One hacker (H4) experienced challenges switching from Python 2 to Python 3, while the other (H11) did not have practical knowledge of pwntools and had to frequently consult the documentation.

Two hackers (H5, H8) attributed a *possibility* of mistake occurrence due to **fatigue** or exhaustion resulting from the time-consuming exploitation process, mentioning lack of sleep. H8 stated after nearly three hours of work:

> *I'm officially at the point in the challenge where I've been looking at it long enough that I'm starting to make stupid mistakes*

6.2. Fixing Mistakes

During debugging mistakes, we observed all hackers **setting expectations** of a certain outcome:

> *I hope this works like this because it should work like this* (H4)

If the anticipated outcome was not met, *confusion* arose, leading hackers to generate and **test multiple hypotheses** — either mentally or through debuggers — to comprehend the issue. We observed this behavior for all eleven hackers. Besides adding `print` statements to the code, hackers often employed standard **exploit debugging techniques**, for example, sought actions with observable side effects (H2, H4, H8), changed code patterns to obtain information (H4), attempted jumps to subsequent function instructions (H4). During the *debugging phase* all hackers, except one, extensively used **online resources** for guidance, mostly conducting searches in Google, reading source code for library functions, or viewing manual pages for certain commands in the Linux terminal. **Trial and error** became more prevalent during the *implementation phase*, especially when initial fixes proved ineffective. We noticed that all hackers relied on trial and error and were merely brute-forcing or iteratively adjusting strategies until they achieved a successful resolution.

Often, while constructing an exploit, hackers (H1, H2, H4-7) **anticipated** that it would not work flawlessly right away and foresaw issues with its functionality:

> *We're probably going to corrupt RSP, aren't we?* (H7)
> *I think I missed something in the ROP chain* (H1)

Some hackers were building and testing the code incrementally, mitigating issues as they emerged. H2 recommended to his audience:

> *This is why you develop things piecemeal, because you make tiny mistakes...*

6.3. Avoiding Mistakes

Some hackers (H1, H2, H4, H8, H9) attempted to minimize the risk of making mistakes and avoid confusion by being **organized** and **exercising good** command line and coding **practices**. For example, H8 shared:

> *I always put my compile and run commands on the same line, so that I don't make that mistake*

Additionally, several hackers (H4, H8) **future-proofed** their code by considering potential changes in the system and environment, and one hacker (H1) admitted the need to use a linter after spending some time on fixing typographical errors in exploit code.

7. Hacker Dynamics—Who Made Mistakes?

Successful hacking not only relies on technical knowledge but also strategic finesse. Our hackers exhibited multifaceted behavior as they delved into the complex and time-consuming exploitation process. Next, we discuss the

hackers' efficiency-increasing approaches, their perceptions of tools used during exploitation, and the dynamics of online interactions to highlight their deliberate choices, their adaptive approaches, and the overall collaborative nature of problem-solving enabled via streaming.

7.1. Efficiency Increasing Strategies

We identified several optimization patterns employed by the hackers, especially during the debugging phase (Section 5) to maximize their *efficiency*. H4, H5, H6, and H8 attempted to **avoid time-intensive** and unnecessary **actions**. Depending on the task, they tried to brute-force solutions, automate repetitive tasks, and leverage custom scripts to expedite exploitation. Notably, several hackers avoided a sub-optimal approach to exploit writing (H7), excessive reusability of reference code (H6), or inefficient algorithms (H5), seeking a "nicer" and "more intelligent" way.

We noticed several instances when hackers were identifying and **filtering out not-so-relevant information** for their task (H1, H4-8). In addition, six hackers **relied on past experience** by reusing templates or previously written code (H2), looking for commonalities with the challenges they had already solved (H4, H5), and using tricks that they learned from CTFs to resolve problems (H1, H8). A few hackers recounted their experience with occasional flaws and misconfigurations of CTF setups that helped them solve challenges in the past and imparted insights about tactics and techniques they employed during CTF competitions (H4, H8). Multiple instances occurred where hackers retrospectively recognized the **potential improvements** they could have incorporated into their exploit by using appropriate tools to increase efficiency (H6-8). H8 recounted:

> *I remember seeing the pwntools one [pwntools option], and I'm like: Wow! I wasted a lot of time on that challenge!*

Six hackers explored **shortcuts** or **novel ideas** in an attempt to optimize the exploitation process:

> *Let's just try this for speed... How did that work? I'm not gonna question that* (H5)

7.2. Navigating Tools and Tactics

In addition to using standard tools, seven hackers (H1-6, H8) opted for customized solutions or developed their own applications, helping them to craft or debug exploits. Several demonstrated the capability to use a variety of **techniques** to perform a single task, for example, to check an address or calculate a position on the stack. Nine hackers relied on the pwntools library for creating their exploits:

> *[pwntools] should be in any hacker's arsenal because it makes prototyping things so much faster* (H8)
>
> *They have all of this magic stuff baked in, it's beautiful!* (H6)
>
> *It [pwntools] just gives you so many nice little bits and pieces for running binaries and interacting with them* (H7)

To perform ROP attacks to bypass security mechanisms, e.g., non-executable stack (NX), hackers often used command-line tools, like One Gadget, to search binaries for usable instructions. Despite the high usability of these tools, H1 mentioned issues with One Gadget as well as the bugs in pwntools that he could not explain.

Hackers unanimously used GDB (GNU Debugger framework) extensions, such as pwndbg, peda, and gef, except streams created in 2014-2015 and cases where the hacker had to employ plain GDB on the server. Some hackers explicitly stated their preferences:

> *I find GDB to be pretty unusable without peda or an extension of some kind* (H8)
>
> *GDB's API is so bad... It's good that I have pwndbg* (H1)

Apart from problems with GDB, some hackers criticized the cryptic and confusing GLIBC source code (H8) and the lack of clarity in Radare documentation (H11). The **shortcomings** within multiple tools impeded the hackers' efficiency in the exploitation process.

7.3. Hackers' Streaming Interactions

Collaborators and individualists. Broadly, we observed that hackers had two online behaviors while streaming, namely collaborators ($n = 4$) and individualists ($n = 7$). Collaborative hackers considered the streaming event as an opportunity to *socialize and engage with the audience*. We noted instances ($n = 18$) where hackers received advice that helped them resolve their mistakes or saved considerable debugging time, underscoring *collaboration* as a potent tool for refining and enhancing the efficiency of the exploitation process. On the other hand, hackers with an individualist mindset used their stream time to focus on solving the challenge, occasionally attending to the streaming platform's chat feature to answer viewer questions or ask for help. Only three hackers didn't pay any attention to viewers.

Educators. Nine hackers stood out as educators by answering questions, drawing and demonstrating examples, or explaining concepts. At times (H2, H6, H8), *educating* others helped hackers realize mistakes or clarify strategy, as H8 stated:

> *I don't think that would work, now that I'm talking it through*

Some hackers (H3, H5, H6-8) demonstrated specific behaviors in response to audience demands. For example, H3 realized the beginner level of his viewers and attempted to explain complex concepts in "a very simple way." Some hackers offered clarifications to ensure the audience understood their actions. H8 exemplified these **streaming demands** by stating:

> *I'm trying to show good practice since I am showing off to the public*

Learning on-the-go. On the other hand, hackers (H2, H4, H6, H8) often admitted a lack of knowledge or experience but at the same time demonstrated curiosity and perseverance and took time to *learn new things* during the stream and *share knowledge* with the audience. H2 shared:

> *...this is part of why I wanted to do this video series. It's very easy to read some of these walkthroughs and just think that, like, these people are brilliant, they just do X, Y, and Z, and they get the flag, but oftentimes you have a lot of false starts*

down a lot of paths, but in the meantime, you're learning a lot of interesting things

7.4. Emotions and Attitudes

Ten hackers displayed a certain degree of **frustration** or annoyance, usually during the debugging phase of the mistake-solving process. We identified five primary causes of frustration: repeatedly misreading binary code due to distractions (multitasking), grappling with "weird" program behavior, frequent occurrences of typographical errors or minor mistakes, the necessity for extensive debugging, or simply misunderstanding binary and consistently mistaking the approach. The level of irritation usually increased as the problem persisted, and a hacker felt an absence of progress. Depending on the hacker's style, frustration could manifest as a **negative self-talk** (H6-10).

Three hackers originally **underestimated** the difficulty of a challenge, later admitting they were wrong. As H5 stated:

> *Well, you think this problem is gonna be easy and it so isn't*

At times, to circumvent challenging parts or to avoid time-consuming tasks, hackers relied on **luck**.

> *Let's just try the magic gadget remotely, maybe we'll get lucky* (H8)
>
> *[...] just cross our fingers and hope we get a string in the memory* (H7)
>
> *I just put tons of crap on the stack and hoped that we get a zero there and we did* (H5)

8. Discussion and Recommendations

In this section, we discuss the findings of our research, their alignment with real-world scenarios, and the broader implications they hold. Additionally, deriving inspiration from both our findings and the study of researchers' hacking experiences, we advocate for specific enhancements in hacking strategy, tools, and educational materials.

Leveraging YouTube for cybersecurity research. In our study, we collected and analyzed a unique dataset of YouTube videos to investigate mistakes made by hackers during the hacking process. This approach offered a visual representation of the entire exploitation process, enabling a thorough examination of hackers' strategies, tools, and decision-making in their natural environment, without artificial constraints. Remote observation of hackers' activities via screencasts eliminates geographical barriers, facilitating data collection and analysis. Additionally, YouTube videos afford flexibility in analysis, enabling researchers to review, pause, rewind, and annotate recordings as needed. Finally, YouTube data serves as a rich source of publicly available artifacts. By showcasing the value of this methodology, we hope to inspire further exploration of YouTube data in cybersecurity research endeavors.

Take the time for exploit strategization. Second only to *Implementation* mistakes, *Strategy* mistakes had a significant impact on total exploit time (Section 4). Our measurements of these mistakes were conservative since we couldn't measure off-screen and thinking time. We believe this indicates that strategy could have an even bigger impact on hacking time than measured. With this in mind, we find that dedicating time to craft an efficient and accurate strategy can be more beneficial than patching together an incomplete one.

Additionally, we found that inefficiencies in hacking processes arise from rushing without a detailed understanding of a problem. This lack of understanding can be caused or exacerbated by distractions like stream chat, overall fatigue, failure to automate tasks, and overall strategy unawareness.

Software engineering similarities. We found that phases of exploit creation had conceptual and practical overlaps with general software engineering. Our cost-impact analysis (Section 5) reveals that, on average, hackers dedicate 40% of their time to rectifying mistakes within their exploits, with some cases reaching up to 75%. These findings align with works in the area of software engineering, where the majority of engineers dedicate a similar amount of time to debugging [17, 30].

IDEs are very useful for identifying and fixing programming mistakes like syntax errors in general software development. However, in exploit creation, their benefits might not be as pronounced as the size of code that an exploit requires is usually much lesser than that of a software engineering project. This is reflected in the fact that *Programming* errors (such as runtime exceptions or syntax errors) while being the second most frequent mistake in our dataset, were fixed faster than any other mistake type.

Memory-focused tool improvements. In addition to memory operations being the most time-consuming mistakes (Section 4), debugging was the most time-consuming phase of fixing those mistakes (Section 5). Fixing memory-related mistakes required intricate tooling and additions to base debuggers to make root-cause analysis easier. We find that these tools still lack meaningful ways to visualize and inform hackers of the layout and state of memory. Memory-focused improvements in both debuggers and exploitation libraries, like pwntools, can make mitigating and solving these mistakes easier.

Improvements in education design. We found that exploit *implementation* mistakes, notably errors during writing or leaking memory, significantly impacted the efficiency of the hacking process, taking the longest time to rectify, followed by errors in *strategy* selection (Section 4). Mistakes related to *programming* and *tool* usage were addressed quickly and presented less concern. Our finding aligns with recent studies [6, 22], which highlighted that memory management programming mistakes were the most problematic among undergraduate computer science students. We believe that greater emphasis should be placed on addressing this challenging aspect when devising the curriculum for security courses at universities. In particular, it is crucial to improve understanding of how software interacts with memory by focusing on concepts such as the memory layout of a running executable and processes involved in memory

access to retrieve and store data. Another important aspect involves strengthening comprehension of pointer operations in memory-unsafe languages such as C/C++, as these operations are fundamental to memory manipulation and exploitation.

Future directions. In our study, we applied a manual approach to investigate and analyze mistakes intrinsic to the hacking process. While this method offers in-depth insights, its effectiveness can be significantly enhanced by incorporating AI techniques. Utilizing AI algorithms and machine learning can automate mistake detection and identification, offering scalability and enhancing analysis efficiency. Future work could explore the disparities between experts and novices in comprehending mistakes and improving educational methods and training programs for more effective learning and performance-enhancing strategies. We encourage other researchers to explore these avenues for a deeper understanding of expertise development and mistake interpretation in cybersecurity tasks and across disciplines.

9. Conclusions

In our work, we aimed to investigate human mistakes in the vulnerability discovery process. We leveraged 30 YouTube screencasts featuring 11 hackers addressing binary exploitation challenges. Utilizing YouTube data allowed us to capture and annotate a comprehensive course of actions of hackers in terms of mistakes they made and technical problems they encountered while solving challenges. Through detailed technical and qualitative analyses, we examined the origins, types, and impacts of these mistakes. We found that *Implementation* (memory operations) and *Strategy* are the most time-consuming types of mistakes. After noticing a mistake, hackers spend most of their time debugging and fixing it. Even though *Technical Issues* were rare (6% of all problems), they could have a significant impact on hackers' efficiency. Time lost on rectifying mistakes averaged 40% of total hacking time. Our findings have implications for shaping educational programs, refining security analyst approaches, and enhancing tool development practices.

Acknowledgment

We thank our anonymous shepherd and reviewers for their constructive feedback that helped us to improve our work. Additionally, we appreciate the creators who shared their hacking streams on YouTube, which provided us with valuable data for this research.

This work is sponsored by and related to the Department of Navy Award N00014-23-1-2563 issued by the Office of Naval Research (ONR). This work has also received funding from the Department of Defense Grant No. H98230-23-C-0270 and National Science Foundation (NSF) Awards No. 2232911, 1663651, 2247954, 2146568, and 2232915. The material is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement FA8750-19-C-0003, and work supported by DARPA and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract N66001-22-C-4026. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ONR, DARPA, or NIWC Pacific.

References

[1] OverTheWire: Wargames. https://overthewire.org/wargames/. (Accessed on 12/6/2023).

[2] picoCTF. https://picoctf.org/. (Accessed on 12/6/2023).

[3] pwnable.kr. https://pwnable.kr/. (Accessed on 12/6/2023).

[4] Security creators. https://securitycreators.video/. Accessed: 10/6/2023.

[5] Google Vulnerability Reward Program, 2023. https://bughunters.google.com/about/rules/6625378258649088/google-and-alphabet-vulnerability-reward-program-vrp-rules.

[6] Majed Almansoori, Jessica Lam, Elias Fang, Adalbert Gerald Soosai Raj, and Rahul Chatterjee. Towards finding the missing pieces to teach secure programming skills to students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 973–979, 2023.

[7] Aitor Apaolaza, Simon Harper, and Caroline Jay. Understanding users in the wild. In *Proceedings of the 10th international cross-disciplinary conference on web accessibility*, pages 1–4, 2013.

[8] Victoria A Banks, Katherine L Plant, and Neville A Stanton. Leaps and shunts: designing pilot decision aids on the flight deck using rasmussen's ladder. *Contemporary ergonomics and human factors*, 2020.

[9] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. Air Force Institute of Technology, 2012.

[10] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2765–2782, 2022.

[11] Sloane C Burke and Shonna L Snyder. Youtube: An innovative learning resource for college health education courses. *International Electronic Journal of Health Education*, 11:39–46, 2008.

[12] John W Coffey et al. Ameliorating sources of human error in cybersecurity: technological and human-centered approaches. In *The 8th International Multi-Conference on Complexity, Informatics, and Cybernetics, Pensacola*, pages 85–88, 2017.

[13] david942j. OneGadget. https://github.com/david942j/one_gadget/blob/master/README.tpl.md. (Accessed on 12/6/2023).

[14] Kerstin Duschl, Martin Obermeier, and Birgit Vogel-Heuser. An experimental study on uml modeling errors and their causes in the education of model driven plc programming. In *2014 IEEE Global Engineering Education Conference (EDUCON)*, pages 119–128. IEEE, 2014.

[15] David Embrey. Understanding human behaviour and error. *Human Reliability Associates*, 1(2005):1–10, 2005.

[16] Giovanni George, Jeremiah Kotey, Megan Ripley, Kazi Zakia Sultana, and Zadia Codabux. A preliminary study on common programming mistakes that lead to buffer overflow vulnerability. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1375–1380. IEEE, 2021.

[17] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[18] Robert L Helmreich. Managing human error in aviation. *Scientific American*, 276(5):62–67, 1997.

[19] Kenji Itoh, Noriyuki Yamamoto, and Henning Boje Andersen. Assessing task complexity by use of rasmussen's decision ladder: Model and its application to recovery from healthcare adverse event. In *Conference Proceedings Actes du congrès*, page 436, 2017.

[20] Anthony R Jansen, Alan F Blackwell, and Kim Marriott. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, 35(1):57–69, 2003.

[21] Daniel P Jenkins, Malcolm Boyd, and Chris Langley. Using the decision ladder to reach a better design. In *Ergonomics Society Annual Conference*, 2016.
[22] Monika Kaczorowska. Analysis of typical programming mistakes made by first and second year it students. *Journal of Computer Sciences Institute*, 15, 2020.
[23] Jeremy Kirk. A better reason not to use huawei routers: Code from the '90s.
[24] Gary A Klein. *Sources of power: How people make decisions*. MIT press, 2017.
[25] LiveOverflow. Youtube security scene. *Phrack magazine*, 10(70):0x0f, 2021.
[26] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. Code, camera, action: How software developers document and share program knowledge using youtube. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 104–114. IEEE, 2015.
[27] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.
[28] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on human-computer interaction*, 3(CSCW):1–23, 2019.
[29] Neelam Naikar. *A comparison of the decision ladder template and the recognition-primed decision model*. Defence Science and Technology Organisation Fishermans Bend, Australia, 2010.
[30] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25:83–110, 2017.
[31] Tommy Pollock. Reducing human error in cyber security using the human factors analysis classification system (hfacs). 2017.
[32] Jens Rasmussen. Human data processor as a system component bits and pieces of a model. 1974.
[33] Jens Rasmussen. Outlines of a hybrid model of the process plant operator. In *Monitoring behavior and supervisory control*, pages 371–383. Springer, 1976.
[34] Jens Rasmussen. *Information processing and human-machine interaction. An approach to cognitive engineering*. North-Holland, 1987.
[35] James Reason. *Human error*. Cambridge university press, 1990.
[36] Hendrig Sellik, Onno van Paridon, Georgios Gousios, and Maurício Aniche. Learning off-by-one mistakes: An empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 58–67. IEEE, 2021.
[37] Rebecca Smith and Scott Rixner. The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 538–544, 2019.
[38] Chareen Snelson. Youtube across the disciplines: A review of the literature. *MERLOT Journal of Online learning and teaching*, 2011.
[39] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*. Sage publications, 1990.
[40] David R Thomas. A general inductive approach for qualitative data analysis. 2003.
[41] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 109–126, 2020.
[42] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.
[43] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.
[44] Laurie A Werner and Charles E Frank. A new look at security education: Youtube as youtool. *Information Systems Education Journal*, 8(34):n34, 2010.
[45] David Woods, Leila J Johannesen, Richard I Cook, and Nadine B Sarter. Behind human error: Cognitive systems, computers, and hindsight. 1994.

Appendix A.

A.1. Digging Into Implementation Mistakes

Leak. 20% of the *Implementation* mistakes occurred while hackers attempted to read a memory value. Hackers were able to detect these issues by either obtaining the wrong output or observing a segmentation fault (SIGSEGV) in a debugger. The root causes of these mistakes included using a wrong format string specifier, miscalculating the offset to leak a library address, or computing an address from leaked values incorrectly. Figure 9 shows an example of an incorrect computation of the address of `system` from leaks where the hacker adds values instead of subtracting the offset from the write address. In other cases hackers were using erroneous exploit logic while performing pointer operations for leak such as reading an address from the GOT table or returning a value from a function.

The mistakes took from 1 up to 21 minutes to resolve with the median time of 7 minutes.

```
1  ...
2  write_offset = XXXXXX
3  p = pwn.process('./vuln')
4  p.recvuntil('write: ')
5  write_addr = int(p.recvline().strip(), 16)
6  system_addr = write_offset + write_addr
7  bin_sh_addr =
8  int(p.recvline().split(' ')[1].strip(), 16)
9  ...
```

Figure 9: incorrectly computed the address of system from the leaks.

Write what. A significant portion of all *Implementation* mistakes (72%) encountered during exploit writing occurred while crafting payloads to write into memory (so-called `write-what-where`) where streamers had to realize modification of memory at a controlled location.

We recognized 18 problems that hackers experienced while writing a specific value into memory with eight of them stemming from mistakes in payload. Two mistakes were related to failing function argument requirements by passing wrong values in the payload. One example of such a mistake where a hacker forgot to assemble the shellcode and passed it as a text is shown in Figure 10. One person forgot to account for requisite conditions before sending the exploit, and the other attempted to call `system` with a pointer instead of a string containing "sh", which caused the program termination. Two payloads did not meet the necessary constraints to exploit. In the first case hacker introduced illegal characters in the payload, that prevented the exploit from working properly. The other case pertains to the failure to set a specific register value when manipulating control flow. The last two mistakes were related to using the

wrong heap object to call a function pointer and to write a payload in `use-after-free` challenges.

```
1  ...
2  payload = pwn.shellcraft.amd64.linux.sh()
3  p.recvuntil('>')
4  p.sendline(payload)
5  ...
```

Figure 10: forgot to compile the assembly instructions to shellcode.

Three hackers experienced program failure as the result of a prohibited overwrite. In two cases, stack cookie and `stdout` were overwritten with illegal values. In the third case, the streamer overwrote a pointer with a constant leading to a segmentation fault after dereferencing. Figure 11 shows this mistake.

```
1 list_items('yA' + p32(0x1773)
2     + p32(elf.got['puts']) + p32(0) + p32(0))
```

Figure 11: overwrote a pointer with a constant, the program crashes when trying to dereference it.

Three mistakes occurred due to the use of the wrong address in the payload. One hacker misunderstood the write-what-where condition and used the GOT address of `system` instead of the PLT, but proceeded with writing and testing other functionality of exploit code, and only noticed the error much later when he was explaining the approach to chat. The other hacker committed a similar mistake returning to the address in the GOT where it should have been the PLT, which resulted in a segmentation fault. In the third case, the hacker mistook the address of `main`, but rectified that by investigating output in the debugger.

Two people committed mistakes while constructing the ROP chain by forgetting to include certain gadgets.

We also identified three cases of stack misalignment issues. Two of them occurred as a result of running exploit and were corrected by streamers within 1-2 minutes. Moreover, one of the hackers initially confirmed misalignment and attempted to correct it but failed. The third case was secondary to the wrong function analysis (*Miscellaneous*).

The median solving time for this type of mistakes was 7 minutes, but two issues took more than 50 minutes to solve and hackers spent a significant amount of this time trying to understand the problem.

Write where. 18 issues occurred due to writing to the wrong location in memory with 13 cases of mistakes in the offset to overwrite a pointer. Mistakes included using incorrect pointer operations (six cases) and miscalculating the size of the offset (three cases). The rest were unique scenarios of adding an offset to the wrong gadget in the ROP chain, calculating the offset based on the wrong function address, miscalculating a pointer position on the stack by using a memory address of a variable instead of a pointer to this address, and not accounting for the size of the writable area. Notably, the last mistake occurred at the beginning of writing an exploit and did not appear until the very end, thus lasting more than 50 minutes from origin to mitigation.

Figure 12 demonstrates a mistake in the offset where the author overwrites saved `eip` register instead of the saved `ebp` to perform a stack pivot.

```
1  ...
2  saved_ebp = stack_leak - 28
3  target_on_heap = heap_leak + 24 + 8 + 4 + 4
4  p += p32(saved_ebp)
5  p += p32(target_on_heap)
6  # Main's new saved EIP
7  p += p32(0xXXXXXXXX)
8  # Main's new saved EBP
9  p += 'CCCC'
10 ...
```

Figure 12: overwrote saved EIP instead of the saved EBP to perform a stack pivot (lines 4 and 5 are in the wrong order).

We identified three mistakes related to using the wrong address of overwrite. In one case, a hacker attempted to overwrite a pointer in the PLT instead of the GOT, which crashed the program. In the other, he pinpointed an address in the PLT while demonstrating to the audience that the GOT table was writable. In the third instance, a streamer mistook the address of a function he tried to overwrite. An example of a wrong overwrite is shown in Figure 13.

Two people solving the same challenge with out-of-bounds memory write vulnerability made mistakes by using incorrect pointer shift operation and failing to overwrite at the proper location.

This type of mistakes covers a wide range of time from 1 to 52 minutes averaging to 13 minutes per mistake. The median time to resolve was 7 minutes.

```
gdb-pedas$ x/16wx 0xXXXX3d0
0xXXXX3d0 <puts@plt>: ...
0xXXXX3e0 <system@plt>: ...
...
$ nc 2018shell.picoctf.com XXXX
I'll let you write one 4 byte value to memory.
Where would you like to write this 4 byte value?
0xXXXX3d0
...
```

Figure 13: tried to overwrite the PLT instead of the GOT.

Write what/where. We detected four instances of mistakes where streamers were overwriting the wrong location with an incorrect value. In two cases, in addition to the miscalculated offset, the payload contained either an incorrect order of bytes or a misplaced function address and arguments. Figure 14 contains a code snippet for the latter. In the third scenario, the hacker misconfigured the ROP chain by forgetting to pop a register value from the stack, but also did not account for the stack space, thus moving to a non-writable section of the binary. In the last case, the hacker did not account for the moving stack when calculating the offset for the format string vulnerability and failed to satisfy the constraints for the exploit.

Hackers spent an average of 11 minutes to resolve these issues, with a total time ranging from 5 to 18 minutes.

```
$ python -c "import struct; print 'AAAA' * 24
+ struct.pack('<L', 0xDEADBEEF)
+ struct.pack('<L', 0xDEADCODE)
+ struct.pack('<L', 0xXXXXXXXX)*27" | ./vuln
```

Figure 14: used incorrect offset and order of a function address and arguments.

A.2. Abandoning Issues

There were 23 abandoned issues in total with a cumulative loss of time of more than 2 hours. Issues included all 14 *Strategy* mistakes, four *Programming/Miscellaneous*, two *Tooling*, and three *Technical Issues*. We observed abandonment occurred during different phases.

Six *Strategy* mistakes never explicitly manifested and were realized on a mental level. Hackers spent from 9 to 23 minutes before forsaking their designs, excluding one case where a hacker attempted to overwrite the GOT table entry of `free` with `system` but the payload was too long for this. As a consequence the hacker discarded the mistake within seconds. Other solution design mistakes occurred when hackers attempted to implement or test their idea leading to a program crash or unintended output. Hackers spent from 1 to 13 minutes on these problems before abandoning them.

Two user input errors resulted from accidental mistyping and miscalculating. One was quickly corrected after detection by viewers, the other led to failure in testing exploit and caused a streamer to restart the sub-task he was focusing on. Two other instances involved the incorrect setting of an environment variable and reading `stderr` from pwntools, which hackers abandoned after 40 seconds and 4 minutes respectively, by finding alternative options.

Two cases of *Tooling* mistakes related to hackers failing to find suitable memory addresses in GDB while designing their solutions either due to using the wrong command or forgetting about binary permissions. Both cases were abandoned shortly after the failure.

In three cases, streamers experienced technical difficulties related to utilized software or missing libraries for running 32-bit applications. While two problems were abandoned at an early stage with the hacker finding another approach to proceed with the challenge, the third issue, involving an unsuccessful attempt to initialize register in pwndbg, took 20 minutes before eventually being abandoned.

It is worth noting that despite abandoning some problems, hackers were able to complete all challenges.

A.3. Study Artifacts

Our study artifacts can be accessed at the following URL: https://github.com/sefcom/hackers-mistakes-paper-public-access.

Issue	Type	Sub-type	Sub-sub-type	Count
Mistake	Implementation	Leak	Wrong computation of addr from leaks	1 (1%)
			Wrong exploit logic for leak	4 (3%)
			Wrong format str specifier	3 (2%)
			Wrong offset for leak	2 (2%)
			Total	10 (8%)
		Write What	Mistake in payload	8 (6%)
			Mistook address	3 (2%)
			Rop chain	2 (2%)
			Stack misalignment	2 (2%)
			Wrong overwrite	3 (2%)
			Total	18 (15%)
		Write Where	Mistake in offset	13 (10%)
			Mistook addr of overwrite	3 (2%)
			Pointer shift operations for write	2 (2%)
			Total	18 (15%)
		Write What Where	Mistake in payload / Wrong offset	1 (1%)
			Rop chain / wrong offset	1 (1%)
			Wrong overwrite / wrong offset	2 (2%)
			Total	4 (3%)
		Total		50 (40%)
	Programming/Misc	Programming	Exploit logic error	13 (10%)
			Runtime exception error	19 (15%)
			Syntax, compiler optimization	2 (2%)
			Total	34 (27%)
		Miscellaneous	Analysis of wrong function	1 (1%)
			Configuration error	1 (1%)
			User input error	5 (4%)
			Total	7 (6%)
		Total		41 (33%)
	Strategy	Failed to trigger exploit		1 (1%)
		Faulty payload		10 (8%)
		Incompatible exploit		3 (2%)
		Total		14 (11%)
	Tooling	Debug oversight		2 (2%)
		Process control / debug config		4 (3%)
		Wrong breakpoints		1 (1%)
		Wrong command		5 (4%)
		Total		12 (10%)
	Total			117 (94%)
Tech issue	Application problem			3 (2%)
	Missing dependencies			2 (2%)
	Rand. prog. behavior			2 (2%)
	Total			7 (6%)

TABLE 3: Number of mistakes and technical issues identified in recordings.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper explores the types of and reasons for mistakes made by hackers during the hacking process by analyzing 30 YouTube recordings of 11 content creators solving CTF-style challenge. The authors identify 117 mistakes categorized in four types and highlight factors contributing to these mistakes, debugging strategies employed by hackers, tool limitations, costs associated with mistakes, and potential improvements in education design and tool development.

B.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper provides a valuable contribution to the field by exploring hackers' mistakes and reasoning made during the vulnerability identification and exploitation process, moving beyond the focus of prior work on the causes of vulnerabilities during coding.
2) The paper describes a unique and creative approach for security research, utilizing YouTube recordings of content creators solving CTF-like challenges as data source.