PWN The Learning Curve: Education-First CTF Challenges

Connor Nelson
Arizona State University
connor.d.nelson@asu.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

ABSTRACT

We address the pressing need for effective and scalable cybersecurity education methodologies for undergraduate students. While Capture The Flag (CTF) challenges have been instrumental for some learners, for many novices CTF challenges are simply too difficult and too intimidating to be pedagogically effective. By dissecting and individually presenting these concepts through modularized challenges, we introduce a progressive learning curve that allows students to master complex vulnerabilities, even culminating in crafting advanced end-to-end exploits through both userspace and the kernel. Recognizing the learning barriers imposed by debugging and introspection tools, our method uniquely offers self-guiding challenge variants, effectively decoupling problem-solving from tool mastery. Drawing from five years of curating around 400 systems security challenges, this paper details our insights and experiences, emphasizing the pivotal role of an education-first approach over traditional CTFs. Our methodology's success is underscored by our survey results, with an overwhelming majority of participants acknowledging its pivotal role in deepening their cybersecurity understanding. Furthermore, we have successfully leveraged this material as the foundational content for a follow-on vulnerability research course, where freshly-trained students successfully identified 0-day vulnerabilities in real-world software. As a commitment to global education, we make all challenges and accompanying lecture materials discussed herein freely, and easily, accessible to the world.

CCS CONCEPTS

• **Applied computing → Education**; **Interactive learning environments**; • **Security and privacy → Systems security**; **Software and application security**.

KEYWORDS

Capture The Flag, Cybersecurity Education, Binary Exploitation, Challenge Design, Dynamic Challenge Generation

ACM Reference Format:
Connor Nelson and Yan Shoshitaishvili. 2024. PWN The Learning Curve: Education-First CTF Challenges. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630912

1 INTRODUCTION

Modern cybersecurity requires an understanding and integration of several concepts concurrently. For example, a software exploit might involve a race condition which enables a heap use-after-free. This primitive might cause memory corruption that, in turn, can be used to achieve arbitrary code execution through return oriented programming. If the target is a sandboxed process, OS kernel vulnerabilities (for example, a kernel race condition) might then be abused to gain code execution within the context of the kernel. Is it possible to teach all of these concepts to the average undergraduate computer science student? In order to understand this exploit, students must master reverse engineering, race conditions, dynamic allocator misuse, memory errors, return oriented programming, sandboxing, and kernel security.

Despite this complexity, conveying such understanding is critical. Apple will pay up to $2 million for a zero-click remote chain that provides full kernel execution with persistence, includes a kernel PAC bypass, and circumvents the protections of Lockdown Mode [1]. Such exploits fetch this price because they are constantly being sought out, found, and used by bad actors to circumvent the security of people and organizations [30]. Effective education methodologies to instil in students an ability to identify and fix such flaws is critical to the security of the modern software ecosystem.

In this paper, **we implement and discuss a novel methodology for teaching advanced cybersecurity concepts to mastery at scale**. We explore the strategies necessary for students to not only conceptually understand vulnerabilities, but to apply this knowledge by crafting advanced end-to-end exploits from scratch. Furthermore, we address the challenge of achieving this at scale, catering to hundreds or even thousands of students. By the end of a single semester undergraduate course at a large public R1 research university, students—some of whom have previously never seen x86-64 assembly—craft end-to-end exploits which, for example, leverage kernel vulnerabilities to achieve arbitrary code execution in a sandboxed process with modern mitigations in place. Two crucial observations guide the design of our methodology.

First, we observe that **although security concepts must be integrated concurrently, they need not be taught concurrently or even atomically**. We explore individual cybersecurity concepts each in their own module, building them up gradually through a series of challenges. We begin with a basic and simplified problem and gradually introduce more advanced concepts and realistic constraints, progressively weaving through constituent concepts and gently introducing ideas that will be crucial to succeeding in later challenges. By minimizing conceptual jumps between challenges, we aim to flatten the learning curve and make the material more approachable. After the students master several concepts independently, we combine them.

Second, we observe that students **need not learn to apply a concept, while simultaneously learning the often-times**

complex introspection tools necessary to debug and understand why their solution is failing. Students often struggle to appropriately employ tools like gdb and strace for aiding their understanding of the problem. This presents a massive learning barrier, as they have no way to know if their solution is failing because of a bug in their solution or a bug in their understanding of the concept. To solve this problem, we have developed *self-guiding* variants of all challenges. This allows students to understand the concept before moving on to the regular variant, which requires them to use their own analysis tools such as gdb to solve the challenge. This approach helps overcome difficulties of performing introspection and debugging by decoupling them from the learning of the concept itself, and reduces barriers to progress for many students.

Our challenges draw inspiration from Capture The Flag (CTF), but we have observed that while CTFs are valuable, they are not a universal solution in cybersecurity learning: CTF challenges are designed to be challenging, not necessarily educational. Adopting an education-first approach, we created over 400 challenges over five years, tailored for our undergraduate Computer Systems Security course. These challenges have been integrated into a globally accessible educational website, teaching binary exploitation concepts to over 10,000 students. In this paper, we detail our experience designing these challenges and discuss the lessons we have learned in the process: highlighting the design of the challenges, what worked, what didn't, and why. To evaluate our methodology's effectiveness, a survey was administered to students who participated in our course. The results showed that a majority found the challenges instrumental in grasping cybersecurity concepts. Following this, a subsequent course on vulnerability research indicated that students could leverage this foundational knowledge to identify and report vulnerabilities in real-world software. As a commitment to global education, we make all challenges and accompanying lecture materials discussed herein freely, and easily accessible, to the world, available at https://pwn.college/.

2 RELATED WORK

Researchers have explored the use of hacking as a pedagogical tool to teach computer science and cybersecurity concepts. In [5], Bratus defines hacking by "the ability to question the trust assumptions in the design and implementation of computer systems rather than any negative use of such skills." Through the hacker lens, students are able to perceive a cross-layer view of computer systems. This approach encourages students to think critically and creatively about computer systems. By analyzing the failure modes of systems from a hacker's perspective, students can gain a deeper understanding of how the various layers of a system interact and can be exploited.

Applied cybersecurity labs are an effective pedagogical approach for cybersecurity concepts such as SDN Security [23], Reverse Engineering [2], SQL Injection [4], and Android mobile security [16]. Moreover, researchers have proposed using hacking as toy problems to teach various underlying computer science concepts [22]. For example, parsing can be taught through the example of intrusion detection systems, file systems can be taught through forensics and recovering deleted files, and assembly and memory allocation can be taught through shellcode and buffer overflows.

Capture The Flag (CTF) events are popular in the cybersecurity community as opportunities for participants to apply their knowledge to solve complex challenges in a competitive format. The educational merit of hacking events has been acknowledged by the academic community, as evidenced by academic competitions like iCTF [25], picoCTF [8], and CSAW CTF [14] that specifically cater to students and enable them to showcase their abilities and engage in peer competition. CTF events have become an effective way to cultivate the next generation of cybersecurity experts [3, 11] and advance the state of the art in cybersecurity. Increasingly, CTF is being used within academic courses, with many educators incorporating them into their courses as exercises [19] or even making them the focal point of a flipped classroom model [7, 21]. Educators report increased student engagement, and improved self-confidence and motivation among the students [7, 10, 17].

CTF can be an incredibly valuable educational tool, but there are many challenges and pitfalls, especially for novices. While a flag-based system provides instant feedback on the correctness of an answer, it lacks partial flags or credit, making it difficult to gauge progress. This limitation is especially pronounced in challenges intentionally designed to frustrate competitors, which are unsuitable for educational purposes. Instead, a well-designed challenge should guide students through the solution process, as Chung argues [9]. However, a methodology for designing challenges that strike the right balance between being engaging and informative is challenging and has received limited attention in the literature. In this paper, we focus on addressing these limitations.

Vykopal et al. [26] identified three main reasons why beginners may become discouraged: overly difficult challenges, ambiguous challenges, and limited feedback on progress. In [29], the authors aimed to target a novice population by designing challenges that did not require significant prior knowledge (for example, avoiding compiled binaries, which were considered too difficult). However, novices still struggle to self-learn and require guidance and support. For instance, in [20], researchers provided novices with mentors and taught a lecture on the networking tool tcpdump to prepare them for upcoming challenges. The authors of picoCTF found that competitors tend to prefer challenges that require minimal familiarity with the command line or additional tools, and consequently stressed the importance of teaching tooling to students [8]. Similarly, [28] found that while there is no way to fake the knowledge needed in a CTF challenge, the game must be at an appropriate level for the audience. If students have not studied networking and lack experience with the command line, they may experience difficulty and frustration. As a solution, the authors suggest creating tutorials and "level zero" versions of challenges to provide a foundation for students to build upon. This approach allows novices to gradually develop their skills and knowledge, gaining confidence and motivation as they progress through the challenges. In this paper, we expand upon this idea of a "level zero" version of challenges, by providing an entire progression of challenges that build upon each other to ramp up the difficulty and complexity, and continue to build up confidence and knowledge along the way.

In order to gain better insight into how students approach and progress in solving challenges, some tools have been developed which introspects their process [18, 27]. These tools help educators identify challenging tasks and improve educational materials. In this paper, we focus specifically on the process by which we enhance those educational materials, and in particular, enhance the challenges. We take the insight of where a student is struggling and use it to guide the design of predecessor challenges that will help them succeed.

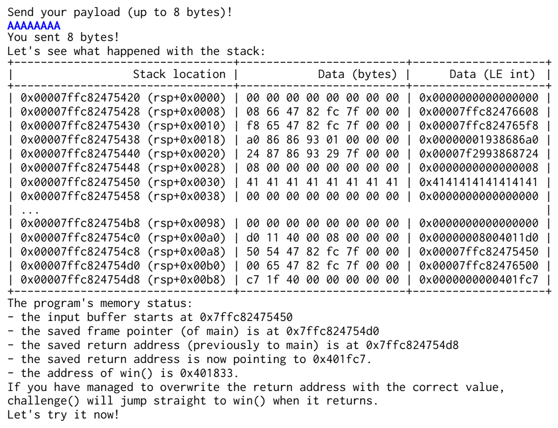
Designing effective challenges can be both a difficult and time-consuming task. Alpaca [12] is a system that uses AI and a database of vulnerabilities to generate challenge scenarios automatically according to user-specified constraints. The generated scenarios include a series of vulnerabilities and exploits that must be overcome to complete the challenge and create virtual machines with these vulnerabilities built-in. Users can specify complexity levels and specific vulnerabilities that must be used. Conversely, SecGen [24] generates challenges by randomly selecting composable modules. In [13], the authors automatically generate variants of reverse engineering challenges in order to prevent cheating. picoCTF also focuses on automatic challenge generation [6], but uses a much more trivial find-and-replace approach against challenge values. In this paper, we focus on templated challenge generation which enables very slight modifications within low level program details. Rather than using different high level vulnerabilities present within a scenario, or focusing only on the task of replacing specific values in a challenge, as done in prior work, our templating methodology is more generic and allows us to easily create a progression of challenges that build upon each other by specifying only the differences between challenges. This provides a significant advantage over the prior work, as it allows our challenges to be much more maintainable, and is what enables challenge progression in the first place.

3 INDIVIDUALIZED CONCEPTS

We split the complex area of binary security into a number of single-concept modules, which we detail in this section. Because Linux-based infrastructure is easier to build and maintain, our challenges explore these concepts in a Linux environment, although analogues can conceptually be created for other platforms as well. Successful solution of a challenge allows students to retrieve challenge-specific secret files, called *flags*, that they can redeem for credit in our institution's Computer Systems Security course.

Shellcode Injection. Shellcoding is the art of injecting code into a program, usually during exploitation, to induce attacker-desired actions. It is traditionally taught by presenting students with a stack buffer overflow and an executable stack, and expecting students to figure out memory corruption, an exploit payload, as well as learning how to debug their exploit all at once. Instead, this module explicitly focuses on the payload by ingesting shellcode and immediately executing it, allowing students to fully focus on learning about shellcode itself. Throughout the module, we gradually introduce additional constraints on this shellcode to force students to reason creatively about accomplishing attacker goals under difficult situations.

Before running shellcode, teaching variants disassemble the ingested payload, after any mutations and constraints are applied, and display it. Additionally, we have recently implemented an automatic breakpoint at the beginning of the shellcode when students start the challenge from a debugger, to ease troubleshooting. While students



```
Send your payload (up to 8 bytes)!
AAAAAAAA
You sent 8 bytes!
Let's see what happened with the stack:
+----------------------------------+-------------------------+--------------------+
|                   Stack location |            Data (bytes) |      Data (LE int)  |
+----------------------------------+-------------------------+--------------------+
| 0x00007ffc82475420 (rsp+0x0000) | 00 00 00 00 00 00 00 00 | 0x0000000000000000 |
| 0x00007ffc82475428 (rsp+0x0008) | 08 66 47 82 fc 7f 00 00 | 0x00007ffc82476608 |
| 0x00007ffc82475430 (rsp+0x0010) | f8 65 47 82 fc 7f 00 00 | 0x00007ffc824765f8 |
| 0x00007ffc82475438 (rsp+0x0018) | a0 86 86 93 01 00 00 00 | 0x00000001938686a0 |
| 0x00007ffc82475440 (rsp+0x0020) | 24 87 86 93 29 7f 00 00 | 0x00007f2993868724 |
| 0x00007ffc82475448 (rsp+0x0028) | 08 00 00 00 00 00 00 00 | 0x0000000000000008 |
| 0x00007ffc82475450 (rsp+0x0030) | 41 41 41 41 41 41 41 41 | 0x4141414141414141 |
| 0x00007ffc82475458 (rsp+0x0038) | 00 00 00 00 00 00 00 00 | 0x0000000000000000 |
| ...                                                                            |
| 0x00007ffc824754b8 (rsp+0x0098) | 00 00 00 00 00 00 00 00 | 0x0000000000000000 |
| 0x00007ffc824754c0 (rsp+0x00a0) | d0 11 40 00 08 00 00 00 | 0x00000008004011d0 |
| 0x00007ffc824754c8 (rsp+0x00a8) | 50 54 47 82 fc 7f 00 00 | 0x00007ffc82475450 |
| 0x00007ffc824754d0 (rsp+0x00b0) | 00 65 47 82 fc 7f 00 00 | 0x00007ffc82476500 |
| 0x00007ffc824754d8 (rsp+0x00b8) | c7 1f 40 00 00 00 00 00 | 0x0000000000401fc7 |
+----------------------------------+-------------------------+--------------------+
The program's memory status:
- the input buffer starts at 0x7ffc82475450
- the saved frame pointer (of main) is at 0x7ffc824754d0
- the saved return address (previously to main) is at 0x7ffc824754d8
- the saved return address is now pointing to 0x401fc7.
- the address of win() is 0x401833.
If you have managed to overwrite the return address with the correct value,
challenge() will jump straight to win() when it returns.
Let's try it now!
```

Figure 1: Memory Errors teaching variant.

can, of course, do both of these things manually, simplifying the debugging process helps avoid frustration.

Reverse Engineering. Reverse Engineering is the process of analyzing a system in order to derive knowledge of its design and implementation. While this often means simply learning to read assembly, in order to truly master reverse engineering, it is important to be able to infer and understand the core design of a target system. This understanding, in turn, is typically used to reason about vulnerabilities in binary code. In this module, however, we free students from the need to reason about vulnerabilities by creating challenges requiring students to understand and invert algorithms that are implemented with (through the course of the module) increasing amount of obfuscation.

Teaching variants display the results of data mutations as they occur, allowing students to double-check their hypotheses. In later levels, teaching variants display the VM state and executed instructions to enhance initial student understanding.

Memory Errors. Memory errors occur when a user is able to corrupt memory they're not supposed to, the result of which can have brutal effects and may enable significant control over a program. To understand this concept requires an understanding of a program's runtime as it uses control flow metadata, common issues surrounding buffer overflows and buffer overreads, as well as the impacts of static and dynamic memory layouts. Over the course of this module, we guide students from a simple overflow into a boolean variable through a number of complex concepts that require students to reason about security mitigations, exploitation in partial-knowledge settings, and other concepts. To reduce the number of concepts they have to consider, we provide a "win" function for students to eventually redirect execution to, rather than forcing them to reason about shellcode.

Teaching challenges in this module display critical memory data, especially control data stored on the stack such as return addresses, canaries, and saved registers, as shown in Figure 1. While this information is accessible through a debugger, having it readily available allows students to focus on understanding the concept first, then understanding the debugger later.

Return Oriented Programming. Return Oriented Programming (ROP) is a *code reuse* attack in which attackers redirect control

```
$ echo -ne "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\t'@\x00\x00\x00\x00\x00<\x00\x00\x00\x00\x00\x00\x00
'@\x00\x00\x00\x00\x00*\x00\x00\x00\x00\x00\x00\x008'@\x00\x00\x00\x00\x00" | /challenge
Received 160 bytes! This is potentially 5 gadgets.
Let's take a look at your chain!
Note that we have no way to verify that the gadgets are executable from within this challenge.
You will have to do that by yourself.
+--- Printing 6 gadgets of ROP chain at 0x7ffeed754998.
| 0x0000000000402709: pop rax ; ret ;
| 0x000000000000003c: (UNMAPPED MEMORY)
| 0x0000000000402720: pop rdi ; ret ;
| 0x000000000000002a: (UNMAPPED MEMORY)
| 0x0000000000402738: syscall ; ret ;
| 0x0000000000000000: (UNMAPPED MEMORY)
Leaving!
$ echo $?
42
```

Figure 2: Return Oriented Programming teaching variant.

```
This challenge can manage up to 1 unique allocations.
[*] Function (malloc/free/puts/read_flag/quit): malloc
Size: 407
[*] allocations[0] = malloc(407)
[*] allocations[0] = 0x5630d7b702c0
[*] Function (malloc/free/puts/read_flag/quit): free
[*] free(allocations[0])
+================+=================+==========+=====================+=====================+
| TCACHE BIN #24 | SIZE: 393 - 408 | COUNT: 1 | HEAD: 0x5630d7b702c0 | KEY: 0x5630d7b70010 |
+================+=================+==========+=====================+=====================+
| ADDRESS        | PREV_SIZE (-0x10)  | SIZE (-0x08)   | next (+0x00)   | key (+0x08)          |
+----------------+--------------------+----------------+----------------+----------------------+
| 0x5630d7b702c0 | 0                  | 0x1a1 (P)      | (nil)          | 0x5630d7b70010       |
+--------------------------------------------------------------------------------------------+
[*] Function (malloc/free/puts/read_flag/quit): read_flag
[*] flag_buffer = malloc(407)
[*] flag_buffer = 0x5630d7b702c0
[*] Read the Flag!
[*] Function (malloc/free/puts/read_flag/quit): puts
[*] puts(allocations[0])
Data: FLAG{...}
```

Figure 3: Dynamic Allocator Misuse teaching variant.

flow not to injected shellcode (for example, because Data Execution Prevention mitigations leave no place for executable shellcode), but to a series of disjoint sets of instructions (termed *gadgets*), with control flow transfers facilitated by `ret` instructions coupled with attacker control of the program stack. Throughout this module, we guide students from first calling a single function, to chaining multiple functions, and gradually through the chaining of ROP gadgets in complex scenarios.

Analogous to Shellcode Injection, teaching challenges disassemble the ROP chain that would be executed before executing it. This is a feature not seen even in popular debuggers, and helps students understand what is happening when their ROP chain executes.

Dynamic Allocator Misuse. In recent years, attackers have bypassed exploitation mitigations by abusing dynamic memory allocators to achieve full control of program memory. This has historically been considered a somewhat secret art, and mostly explored in high-skill CTFs. To keep things tractable for students, this module mostly focuses on the details of the implementation of the `tcache` allocation caching layer of Linux's `glibc`, attacks against which are simpler than the general case of allocator misuse.

The teaching variants display the current state of tcache, including cached allocations and any relevant metadata, as shown in Figure 3. While, again, there are tools that can perform this function, including this in the challenge enables students to tackle the problem head on without any impediments.

Sandboxing. Sandboxing is the practice of isolating code to limit attacker capabilities during exploitation. While many courses discuss this concept within systems security, we have found few that actually introduce labs or challenges which practically explore it. To maximize ease of learning, this module assumes arbitrary code execution has already been achieved (by directly running the shellcode that students have learned to write in the previous module) in order to focus entirely on the sandboxing mechanism itself. The specific sandboxing mechanisms used evolve over the course of the module, exposing students to different sandboxing scenarios.

The teaching variants describe the jailing process as it happens, describing the resources that are available at what point of challenge execution.

Race Conditions. Concurrency errors are some of the trickiest and least-understood software bugs. Because they occur in many different contexts and are dependent on the specifics of the software in question, the teaching of concurrency errors seems to be less common than the teaching of other software errors. Our challenges progress from filesystem-based time-of-check to time-of-use (TOCTOU) attacks into complex mutex issues in memory, allowing students to gradually familiarize themselves with underlying concepts.

As challenges get more intricate, introspecting the program and reason about what impact they may have at what point in execution becomes difficult. Teaching challenges in this module stall the program in critical areas, allowing students to experiment with carrying out their attack without time pressure (and resulting implementation difficulties in scripting fast attacks) before resuming the challenge execution at their leisure.

Kernel Security. The security of operating system kernels is considered to be a very advanced topic. Undergraduate OS courses rarely cover interactions with the kernel in any significant way, and undergraduate security courses rarely talk about the kernel at all (other than its effects of memory allocation and permissions). Our challenge design allows us to not only discuss the kernel, but instill in our students a deep-seated understanding of its security implications. This module has a fundamental effect on students in the sense that it expands their experience from "program" hacking to "system" hacking: the first time a student single-steps *into* (as opposed to *over*) a `syscall` instruction to find themselves debugging *the kernel syscall handler*, their perspective fundamentally expands beyond a typical undergraduate security course.

Teaching variants log outputs of kernel APIs used by the vulnerable code. This allows students to, for example, understand what data the kernel accesses from userspace applications.

4 INTEGRATION CHALLENGE DESIGN

To truly achieve proficiency, students must put the individual concepts learned in the modules above together into an end-to-end understanding of cybersecurity. We created two such integration modules: one focusing on the exploitation of individual *programs* (integrating Shellcode Injection through Memory Errors), and one focusing on the exploitation of multi-component *systems* (integrating *all* of the material). These challenges, despite their complexity, are combinations of prior modules. The teaching variant actions in those prior modules are preserved wherever possible.

Program Exploitation. As with the single-concept challenges, a core tenet of the progression is the gradual, bit-by-bit increase in complexity. Thus, the challenges start with a very simple buffer overflow, and become increasingly complicated from there. The final challenge of this module is a version of the VM from the Reverse Engineering module, reimagined as a realistic just-in-time compiler-based rather than interpreter-based VM. Students discover memory corruption issues within the virtual machine to hijack control flow of the program, then abuse the just-in-time compilation to introduce precisely-crafted shellcode. This sort of end to end exploit is conceptually similar to what modern web browser exploits against actual JavaScript engines look like in the real world.

System Exploitation. The second integration module extends into advanced material, exploring, e.g., the impacts of Race Conditions to enable Dynamic Allocator Misuse in multi-threaded software. These challenges convey two concepts: the chaining of radically different vulnerabilities to achieve a goal impossible with either vulnerability alone, and the importance of knowledge of the interconnectedness of program memory regions during exploitation.

These challenges integrate Kernel Security by reimagining the VM from the Reverse Engineering module as a kernel subsystem (similar to the BPF VM that powers the Linux kernel's network and system call firewalls). The final challenge (again, a series of challenges gradually leads to cumulative increases in complexity) requires students to perform a multi-stage exploit using concepts from all of the prior modules to first race heap operations to achieve arbitrary code execution in a sandboxed userspace program, then pivot to attacking the virtual machine kernel subsystem by racing threads to induce memory corruption in the kernel, get a kernel pointer leak, and achieve arbitrary code execution in the kernel, all in order to finally disable the properly-configured seccomp sandbox (by rewriting relevant kernel data structures that track process sandboxing state), escalate privileges to root (again, by carefully corrupting kernel data structures), and get the flag.

5 DISCUSSION

Smoothing Learning Curves. As challenges are responsible for progressively introducing concepts, it is critical to minimize jumps in difficulty throughout the progression of a module. We can measure difficulty jumps by analyzing discontinuities in challenge solve progressions in terms of the percentage of students solving each challenge. In our experience, when a large drop in solutions exists between one challenge and the next, it means that the more difficult challenge is conveying too many concurrent concepts. These concepts should be broken apart into one or more intermediate challenges which build up to that original difficult challenge.

A representative example of this evolution is our Reverse Engineering module. The Fall 2020 iteration of our course had a massive difficulty jump approximately 70% through the module, resulting in half as many solves as the previous challenge. This occurs when the first virtual machine reverse engineering challenge is introduced. Compared to the prior (much simpler) "crackme" challenges, students were required to understand numerous new concepts concurrently in order to solve the challenge: obfuscation, interpretation of bytecode, and the use of complex data structures. Many students failed to make this jump. To remediate this, we added several conceptually simpler challenges in Fall 2021 to prepare students for this challenge. We did this by initially introducing pieces of the VM just as obfuscation for normal code logic, initially avoiding any sort of interpreter loop (and subsequent handling by the VM of its own control flow) or embedded VM instruction interpreters. Over the course of a number of challenges, we introduced these sub-concepts one by one, making it much easier for students new to reverse engineering to adapt. This resulted in a much smoother challenge progression and dramatically improved student success. As opposed to the 50% drop in solves of the first VM challenge, no challenge in Fall 2021 had a solve drop of more than 10%. At this point, we hit a limit in curve smoothness: in 2022, we added additional intermediate challenges because we felt that obfuscation concepts could be introduced even more slowly, but we did not observe a similar improvement in solution rates over Fall 2021.

Challenge Generation. We developed pwnshop, an open-source (available at https://github.com/pwncollege/pwnshop), template-based system for generating and maintaining our hundreds of individual challenges. This system leverages Jinja [15] as a templating layer on top of C source code. In doing so, we can conditionally include or exclude code, which is crucial for maintainably generating a challenge with just a small difference compared to a previous challenge. This capability enables us to fix a bug in a challenge (for example, disabling output buffering), and having that fix propagate to all challenges that are based on that template. Furthermore, this allows us to generate challenges with randomized values, such as randomized passwords or randomized memory layouts. The benefits of this approach are twofold. First, it allows us to generate more versions of the same fundamental challenge, which can be useful for increasing student's practice with a particular concept (for example, calculating buffer overflow offsets). Second, it allows us to generate challenges that are unique to each student, which can be useful for preventing students from sharing complete solutions with each other. In order to make sure that the challenges are still solvable, we have verify scripts that can solve any version of a challenge, and we use these scripts to verify that the challenges are solvable before they are released to students.

Refining Core CS Concepts. An interesting benefit of our challenges to students is the solidification of earlier Computer Science concepts that students may have previously missed. For example, despite many students having prior experience with process scheduling in our institution's Operating Systems course, many of our students did not seem to truly "get" the concept until they had to exploit process scheduling intricacies to attack programs in the Race Condition module. We observed similar effects with concept missed in students' Computer Organization courses, and with privilege and memory isolation in the Kernel Security module.

6 EVALUATION

Survey. To understand the effectiveness of challenge-based learning and our challenge design, we distributed an IRB-exempt (minimal harm) survey to our students. They survey contained both quantitative (Likert scale) and qualitative (free response) components. Since our challenges are openly and freely accessible to learners around the world, we received a wide variety of responses from 200 total respondents. In this section, we distill these responses into insights about the potential of our proposed learning approach.

We present our Likert question results in Figure 4. Our style of challenge-based learning was popular: 93.1% of students reported liking challenge-based learning and 92.6% reported significant learning from these challenges. Students almost universally appreciated the immediate feedback. Our "teaching-variant" challenges were quite well-received, with 84.8% of students reporting being helped

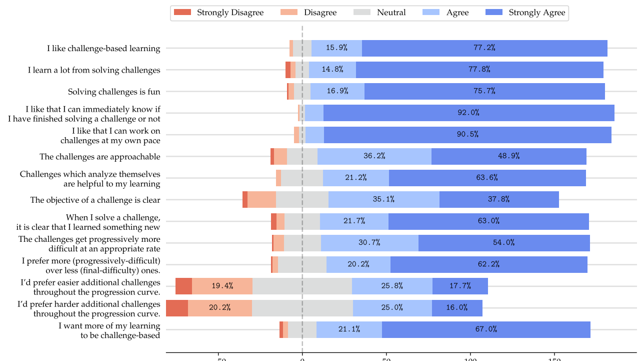


Figure 4: Survey Responses. Total number of participants to respond positively (postive values) and negatively (negative values).

by them. Likewise, 84.7% of students learned something new from solving individual challenges and were satisfied with our challenges' difficulty progression. In fact, 82.2% of students prefer *more* challenges, despite this course obtaining a reputation as the hardest course in our department. There does not appear to be consensus of whether these additional challenges should be on the easier or harder end of the progression. Overall, 88.1% of students would prefer more of their learning to be challenge-based. Questions about individual modules also confirmed the aforementioned refinement of Computer Science concepts.

Another theme was the duality of challenge frustration and the satisfaction derived from solving them. During course design, we often strive to avoid frustration (for example, by smoothing difficulty curves between challenges), but the framing of the responses suggests that some frustration might incentivize students to push through and "defeat" the challenge. In several modules, including Race Conditions, Kernel Security, Memory Errors, and Dynamic Allocator Misuse, students reported that carrying out attacks against related weaknesses helped them develop code resilient to such flaws. Further exploration of this phenomenon could be valuable to the Software Engineering field.

The qualitative responses shed light on some subtleties. While students find challenge-based learning rewarding and engaging, some note that the nature of the class can be "frustrating", and that some challenges are repetitive or lack guidance. Even on these, students enjoy the practical application of skills and the immediate feedback on their progress. Free form responses confirm that students appreciate the progressive difficulty within modules and the self-pacing that they enable. However, they suggest various offering lab time for hands-on, guided assistance with concepts to prevent languishing among the student body. Overall, they believe that challenge-based learning is an effective approach for cybersecurity education but there are clear paths for refined execution.

Follow-on Course. To assess how well the skills acquired from our challenge-based learning approach translated into practical applications, we created a follow-on course on applied vulnerability research. This course was offered to students who had successfully completed all challenges from our preliminary challenge-based learning course. The objective of this course was to give students the opportunity to find and responsibly disclose vulnerabilities in real-world open-source software. A total of 19 students enrolled in this course, forming 5 teams. The teams analyzed a variety of projects including PHP, MuJS, Pillow, Radare2, and a popular Gameboy Advanced emulator. All teams were able to find zero-day vulnerabilities in their respective projects, with several of those vulnerabilities translating into CVEs and a more secure global software ecosystem. Students focused on reverse engineering, automated vulnerability discovery (fuzzing), crash triaging, exploit development, and responsible disclosure.

While we haven't conducted a rigorous evaluation, it appears the skills from our challenge-based learning approach—especially in exploitation processes, reverse engineering, and modern exploit techniques—played a significant role in the course's success. There is a significant breadth and depth of knowledge and skills required to even begin real-world vulnerability research; students must be able to understand threat models, and understand the technical implication of how different exploit primitives can be used together to exploit a vulnerability in order to truly compromise a system's security boundaries. We relied on the fact that all students in this course had, for example, already successfully abused dynamic allocators to leak information, bypass PIE, perform an arbitrary write, and achieve code execution through ROP—and had done so in several contexts, with different constraints, and spent hours debugging these exploit chains to completion. This allowed us to focus exclusively on the more advanced aspects of how to apply these skills to real-world software, with the aid of modern vulnerability research techniques. We are unaware of any other undergraduate university course that has been able to successfully teach these pre-requisite skills to students to the extent that we achieved.

Interestingly, we also discovered that while students were more than capable of producing exploits for the vulnerabilities they discovered, and several did, they were more interested in reporting the vulnerability without a full exploit, so that they could move on to finding more vulnerabilities. This in turn suggests that there are two critical tracks in vulnerability research: the ability to find vulnerabilities, and the ability to exploit them. While in this paper we primarily focused on the latter and happened to achieve the former, further research is needed in order to better understand how this style of material can be optimized explicitly toward finding vulnerabilities in real-world software.

7 CONCLUSION

In this paper, we described a design for a challenge-based cybersecurity curriculum that combines single-concept modules and integration modules comprised of challenges that gradually increase in complexity (conveying sub-concepts one by one) and help guide students through their own solutions. In running cybersecurity courses with these challenges, we observed not only great efficacy in student outcomes in cybersecurity, but a radical refinement of their understanding of core Computer Science concepts. To help the educational community, we have made these challenges, as well as the accompanying lecture material, freely available to the world.

Acknowledgments. This work would not have been possible without the vibrant enthusiasm of the `pwn.college` community, and the generous support of the Department of Defense; thank you.

REFERENCES

[1] Apple. 2023. Apple Security Bounty Categories. https://security.apple.com/bounty/categories/

[2] John Aycock, Andrew Groeneveldt, Hayden Kroepfl, and Tara Copplestone. 2018. Exercises for Teaching Reverse Engineering. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 188–193.

[3] Masooda Bashir, April Lambert, Jian Ming Colin Wee, and Boyi Guo. 2015. An Examination of the Vocational and Psychological Characteristics of Cybersecurity Competition Participants. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[4] Nada Basit, Abdeltawab Hendawi, Joseph Chen, and Alexander Sun. 2019. A Learning Platform for SQL Injection. In *Proceedings of the 50th ACM technical symposium on computer science education*. 184–190.

[5] Sergey Bratus, Anna Shubina, and Michael E Locasto. 2010. Teaching the Principles of the Hacker Curriculum to Undergraduates. In *Proceedings of the 41st ACM technical symposium on computer science education*. 122–126.

[6] Jonathan Burket, Peter Chapman, Tim Becker, Christopher Ganas, and David Brumley. 2015. Automatic Problem Generation for Capture-the-Flag Competitions. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[7] Martin Carlisle, Michael Chiaramonte, and David Caswell. 2015. Using CTFs for an Undergraduate Cyber Education. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[8] Peter Chapman, Jonathan Burket, and David Brumley. 2014. PicoCTF: A Game-Based Computer Security Competition for High School Students. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*.

[9] Kevin Chung and Julian Cohen. 2014. Learning Obstacles in the Capture The Flag Model. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*.

[10] Adrian Dabrowski, Markus Kammerstetter, Eduard Thamm, Edgar Weippl, and Wolfgang Kastner. 2015. Leveraging Competitive Gamification for Sustainable Fun and Profit in Security Education. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[11] Michael H Dunn and Laurence D Merkle. 2018. Assessing the Impact of a National Cybersecurity Competition on Students' Career Interests. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 62–67.

[12] Joshua Eckroth, Kim Chen, Heyley Gatewood, and Brandon Belna. 2019. Alpaca: Building Dynamic Cyber Ranges with Procedurally-Generated Vulnerability Lattices. In *Proceedings of the 2019 ACM Southeast Conference*. 78–85.

[13] Wu-chang Feng. 2015. A Scaffolded, Metamorphic CTF for Reverse Engineering. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[14] Efstratios Gavas, Nasir Memon, and Douglas Britton. 2012. Winning Cybersecurity One Challenge at a Time. *IEEE Security & Privacy* 10, 4 (2012), 75–79.

[15] Jinja. 2023. https://jinja.palletsprojects.com/.

[16] Jean-François Lalande, Valérie Viet Triem Tong, Pierre Graux, Guillaume Hiet, Wojciech Mazurczyk, Habiba Chaoui, and Pascal Berthomé. 2019. Teaching Android Mobile Security. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 232–238.

[17] Kees Leune and Salvatore J Petrilli Jr. 2017. Using Capture-the-Flag to Enhance the Effectiveness of Cybersecurity Education. In *Proceedings of the 18th Annual Conference on Information Technology Education*. 47–52.

[18] Jelena Mirkovic, Aashray Aggarwal, David Weinman, Paul Lepe, Jens Mache, and Richard Weiss. 2020. Using Terminal Histories to Monitor Student Progress on Hands-on Exercises. In *Proceedings of the 51st ACM Technical Symposium on Computing Science Education*. 866–872.

[19] Jelena Mirkovic and Peter AH Peterson. 2014. Class Capture-the-Flag Exercises. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*.

[20] Jelena Mirkovic, Aimee Tabor, Simon Woo, and Portia Pusey. 2015. Engaging Novices in Cybersecurity Competitions: A Vision and Lessons Learned at ACM Tapia 2015. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*.

[21] Mike O'Leary. 2017. Innovative Pedagogical Approaches to a Capstone Laboratory Course in Cyber Operations. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 429–434.

[22] TJ O'Connor, Ben Sangster, and Erik Dean. 2010. Using Hacking to Teach Computer Science Fundamentals. *American Society for Engineering Education, St. Lawrence Section* (2010).

[23] Younghee Park, Hongxin Hu, Xiaohong Yuan, and Hongda Li. 2018. Enhancing Security Education Through Designing SDN Security Labs in CloudLab. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 185–190.

[24] Z Cliffe Schreuders, Thomas Shaw, Mohammad Shan-A-Khuda, Gajendra Ravichandran, Jason Keighley, and Mihai Ordean. 2017. Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*.

[25] Giovanni Vigna, Kevin Borgolte, Jacopo Corbetta, Adam Doupe, Yanick Fratantonio, Luca Invernizzi, Dhilung Kirat, and Yan Shoshitaishvili. 2014. Ten Years of iCTF: The Good, The Bad, and The Ugly. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*.

[26] Jan Vykopal, Valdemar Švábenský, and Ee-Chien Chang. 2020. Benefits and Pitfalls of Using Capture the Flag Games in University Courses. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 752–758.

[27] Richard Weiss, Michael E Locasto, and Jens Mache. 2016. A Reflective Approach to Assessing Student Performance in Cybersecurity Exercises. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 597–602.

[28] Richard S Weiss, Stefan Boesen, James F Sullivan, Michael E Locasto, Jens Mache, and Erik Nilsen. 2015. Teaching Cybersecurity Analysis Skills in the Cloud. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 332–337.

[29] Joseph Werther, Michael Zhivich, Tim Leek, and Nickolai Zeldovich. 2011. Experiences In Cyber Security Education: MIT Lincoln Laboratory Capture-the-Flag Exercise. In *4th Workshop on Cyber Security Experimentation and Test (CSET 11)*.

[30] Google Project Zero. 2021. A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html