Conclusion

Congratulations, you have made it through each major part of *Web Application Security: Exploitation and Countermeasures for Modern Web Applications*. You now have knowledge regarding web application recon, offensive hacking techniques for use against web applications, and defensive mitigations and best practices that can be employed to reduce the risk of your application getting hacked.

In this Conclusion, I will point out some of the key topics discussed in each part. Consider reading through to the end and revisiting any of the following topics if you need a refresher.

The History of Software Security

With proper evaluation of historical events, we can see the origins of modern defensive and offensive techniques. From these origins we can better understand the direction in which software has developed and make use of historical lessons while developing next-generation offensive and defensive techniques. Here are the main takeaways from Chapter 1:

Telephone phreaking

- In order to scale telephone networks, manual operators were replaced with automation that relied on sound frequencies to connect telephones to each other.
- Early hackers, known as "phreakers," learned to emulate these frequencies and take advantage of administrative tones that allowed them to place calls without paying for them.
- In response to phreaking, scientists at Bell Labs developed a dual-tone multifrequency (DTMF) system that was not easily re-

producible. For a long period of time, this eliminated or significantly diminished telephone phreaking.
- Eventually, specialized hardware was developed that could mimic DTMF tones, rendering such a system ineffective against phreakers.
- Finally, telephone switching centers switched to digital and eliminated phreaking risk. DTMF tones remained in modern phones for reverse-compatibility purposes.

Computer hacking

- Although personal computers already existed, the Commodore 64 was the first computer that was user-friendly and budget-friendly enough to cause a massive spread in personal computer adoption.
- An American computer scientist, Fred Cohen, demonstrated the first computer virus that was capable of making copies of itself and spreading from one computer to another via floppy disk.
- Another American computer scientist, Robert Morris, became the first recorded person to deploy a computer virus outside of a research lab. The Morris Worm spread to about 15,000 network-attached computers within a day of its release.
- The US Government Accountability Office stepped in for the first time in history and set forth official laws concerning hacking. Morris went on to be the first convicted computer hacker, charged with a $10,500 fine and 400 hours of community service.

The World Wide Web

- The development of Web 1.0 opened up new avenues for hackers to attack servers and networks.
- The rise of Web 2.0, which involved user-to-user collaboration over HTTP, resulted in a new attack vector for hackers: the browser.
- Because the web had been built on security mechanisms designed for protecting servers and networks, many users' de-

vices and data were compromised until better security mechanisms and protocols could be developed.

Modern web applications

- Since the introduction of Web 2.0, browser security has increased dramatically. This has changed the playing field, causing hackers to begin targeting logical vulnerabilities in application code more than vulnerabilities present in server software, network protocols, or web browsers.
- The introduction of Web 2.0 also brought with it applications containing much more valuable data than ever before. Banking, insurance, and even medicine have moved critical business functionality to the web. This has resulted in a winner-takes-all playing field for hackers, where the stakes are higher than ever before.
- Because today's hackers are targeting logical vulnerabilities in application source code, it is essential for software developers and security experts to begin collaborating. Individual contribution is no longer as valuable as it was in the past.

Recon

Due to the increasing size and complexity of modern web applications, a first step in finding application vulnerabilities is properly mapping an application and evaluating each major functional component for architectural or logical risks. Proper application recon is an essential first step prior to attacking a web application. Good recon will provide you with a deep understanding of the target web application, which can be used both for prioritizing attacks and avoiding detection.

Recon skills give you insight into how a qualified attacker would attack your web application. This gives you the added benefit of being able to prioritize defenses, if you are an application owner. Due to the ever-increasing complexity of modern web applications, your recon skills may be limited by your engineering skills. As a result, recon and engineering expertise go hand in hand:

The structure of modern web applications

- Unlike web applications 20 years ago, today's are built on many layers of technology, typically with extensive server-to-user and user-to-user functionality. Most applications use many forms of persistence, storing data on both the server and the client (typically a browser). Because of this, the potential surface area of any web application is quite broad.
- The types of databases, display-level technology, and server-side software used in modern web applications is built on top of the problems web applications have encountered in the past. Largely, the modern application ecosystem is developed with developer productivity and user experience in mind. Because of this, new types of vulnerabilities have emerged that would not have been possible beforehand.

Subdomains, APIs, and HTTP

- Mastery of web application reconnaissance will require you to know ways to fully map the surface area of a web application. Because today's web applications are much more distributed than those of the past, you may need to become familiar with (and find) multiple web servers prior to discovering exploitable code. Furthermore, the interactions between these web servers may assist you in not only understanding the target application, but in prioritizing your attacks as well.
- At the application layer, most websites today use HTTP for communication between client and server. However, new protocols are being developed and integrated into modern web applications. Web applications of the future may make heavy use of sockets or RTC, so making use of easily adaptable recon techniques is essential.

Third-party dependencies

- Today's web applications rely just as much on third-party integrations as they do on first-party code. Sometimes, they rely on third-party integrations even more than first-party code. These

dependencies are not audited at the same standards as first-party code and, as a result, can be a good attack vector for a hacker.
- Using recon techniques, we can fingerprint specific versions of web servers, client-side frameworks, CSS frameworks, and databases. Using these fingerprints, we may be able to determine specific (vulnerable) versions to exploit.

Application architecture

- Proper evaluation of an application's software architecture can lead to the discovery of widespread vulnerabilities that result from inconsistent security controls.
- Application security architecture can be used as a proxy for the quality of code in an application—a signal that hackers take very seriously when evaluating which application to focus their efforts on.

Offense

Performing application reconnaissance can give an attacker an understanding of insecure surface area from which they can begin developing and deploying exploits to exfiltrate important data or force the application to operate in an unintended way. Part II's key points are as follows:

Cross-Site Scripting (XSS)

- At their core, XSS attacks are possible when an application improperly makes use of user-provided inputs in a way that permits script execution.
- When traditional forms of XSS are properly mitigated via sanitization of DOM elements, or at the API level (or both), it still may be possible to find XSS vulnerabilities. XSS sinks exist as a result of bugs in the browser DOM spec and occasionally as a result of improperly implemented third-party integrations.

Cross-Site Request Forgery (CSRF)

- CSRF attacks take advantage of a trust relationship established between the browser and the user. Because of the trusting nature of this relationship, an improperly configured application may accept elevated privilege requests on behalf of a user who inadvertently clicked a link or filled out a web form.
- If the low-hanging fruit (state-changing HTTP GET requests) are already filtered, then alternative methods of attack, such as web forms, should be considered.

XML External Entity (XXE)

- A weakness in the XML specification allows improperly configured XML parsers to leak sensitive server files in response to a valid XML request payload.
- These vulnerabilities are often visible when a request accepts an XML or XML-like payload directly from the client. In more complicated applications, indirect XXE may be possible. Indirect XXE occurs when a server accepts a payload from the user, then formulates an XML file to send to the XML parser rather than accepting an XML object directly.

Injection attacks

- Although SQL injection attacks are the most widely known and prepared for, injection attacks can occur against any CLI utility a server makes use of in response to an API request.
- SQL databases are (often) guarded well against injection. Automation is perfect for testing well-known SQL injection attacks since the method of attack is so well documented. If SQL injection fails, consider image compressors, backup utilities, and other CLIs as potential targets.

Denial of Service (DoS)

- DoS attacks come in all shapes and forms, ranging from annoying reductions in server performance, all the way to complete interruption for legitimate users.

- DoS attacks can target regular expression evaluation engines, resource-consuming server processes, as well as simply targeting standard application or network functionality with huge amounts of traffic or requests.

Attacking data and objects

- All programming languages have rules in regard to how they handle data and functions in memory. After understanding these rules and their quirks, you can start to build data structures that break out of bounds and cause unexpected operations.
- Certain types of attacks targeting objects, such as mass assignment or serialization attacks, allow an attacker to attack an application simply by understanding and exploiting the boundaries expected in common data formats.

Client-side attacks

- Some forms of attack only target the browser client and, as such, are much harder for a developer to detect and mitigate. In fact, some of these attacks depend on the specific browser being used at the time, meaning they might not be detectable during normal development workflows.
- Tabnabbing, clickjacking, and prototype pollution attacks allow an attacker to steal data and compromise user sessions without requiring the user to walk through a server-side workflow. Because of this, these attacks can be developed offline simply by downloading the client-side code to your local device. This makes exploit development a very simple process.

Exploiting third-party dependencies

- Third-party dependencies are rapidly becoming one of the easiest attack vectors for a hacker. This is due to a combination of factors, one of which is the fact that third-party dependencies are often not audited as closely as first-party code.

- Open source CVE databases can be used to find previously reported, known vulnerabilities in well-known dependencies, which can then be exploited against a target application unless the application has been updated or manually patched.

Business logic vulnerabilities

- Business logic vulnerabilities are vulnerabilities specific to the application an attacker is targeting. These vulnerabilities are more difficult to find because they rely on understanding an application's business logic and finding ways to break out of the intended user flow.
- Because business logic vulnerabilities are vulnerabilities specific to a particular set of *business rules*, these vulnerabilities are very difficult to find with automated tools like static application security testing (SAST), dynamic application security testing (DAST), or software composition analysis (SCA). As a result of this, they are often neglected and live in production applications for some time before being found and either exploited or patched, depending on who discovered them.

Defense

Here are the main takeaways from Part III:

Secure application architecture

- Writing a secure web application starts at the architecture phase. A vulnerability discovered in this phase can cost as much as 60 times less than a vulnerability found in production code.
- Proper security architecture can result in application-wide mitigations for common security risks versus on-demand mitigations, which are more likely to be inconsistent or forgotten.

Secure application configuration

- In addition to writing an application in a secure way, it's important to take advantage of all of the standard security features present in the application's technology stack. Doing so can eliminate a significant amount of manual security effort in replicating common security controls.
- In browser-based applications, security mechanisms are available from the developers of JavaScript, the DOM, and the browser itself. Often these require a bit of configuration to enable. But once enabled, they protect your application against many of the most common types of attack.

Secure user experience

- Protecting the source code and implementation of a web application is a fantastic start, but the most secure web applications also incorporate security in their UI development process.
- Web applications that use security best practices within their UI are much more likely to assist users in adopting best practices like MFA, strong password policies, and more.

Threat modeling

- The threat model is an industry standard and highly valuable tool for discovering and documenting security gaps in an application prior to its development.
- Security gaps found during the threat modeling phase are almost always significantly cheaper and faster to resolve than those found postproduction or after code has been written.

Reviewing code for security

- After a secure application architecture has been decided upon, a proper secure code review process should be implemented to prevent common and easy-to-spot security bugs from being pushed into production.
- Security reviews at the code review stage are performed similarly to a traditional code review. The main difference should

be the type of bugs sought after and how files and modules are prioritized given a limited time frame.

Vulnerability discovery

- Ideally, vulnerabilities would be discovered prior to being deployed in a production application. Unfortunately, this is often not the case. But there are several techniques you can take advantage of to reduce the number of production vulnerabilities.
- In addition to implementing your own vulnerability discovery pipeline, you can take advantage of third-party specialists in the form of bug bounty programs and penetration testers. Not only can these services help you discover vulnerabilities early, but they can also incentivize hackers to report vulnerabilities to your organization for payment rather than selling found vulnerabilities on the black market or exploiting the vulnerability themselves.

Vulnerability management

- Once a vulnerability is found, it should be reproduced and triaged. The vulnerability should be scored based on its potential impact so its fix can be properly prioritized.
- A number of scoring algorithms exist for determining the severity of a vulnerability, with CVSS being the most well known. It is imperative that your organization implements a scoring algorithm. The scoring algorithm you choose is less important than the fact that you use one. Each scoring system will have a margin of error, but as long as it can distinguish the difference between a severe and low-risk vulnerability, it will help you prioritize the way in which work is distributed and bugs are fixed.

Defending against XSS attacks

- XSS attacks can be mitigated at a number of locations in a web application stack: from the API level with sanitization functions, in the database, or on the client. Because XSS attacks tar-

get the client, the client is the most important surface area for mitigations to be implemented.
- Simple XSS vulnerabilities can be eliminated with smart coding, in particular when dealing with the DOM. More advanced XSS vulnerabilities, such as those that rely on DOM sinks, are much harder to mitigate and may not even be reproducible! As a result, being aware of the most common sinks and sources for each type of XSS is important.

Defending against CSRF attacks

- CSRF attacks take advantage of the trust relationship between a user and a browser. As a result, CSRF attacks are mitigated by introducing additional rules for state-changing requests that a browser cannot automatically confirm.
- Many mitigations against CSRF-style vulnerabilities exist, from simply eliminating state-changing GET requests in your codebase, to implementing CSRF tokens and requiring MFA confirmation on elevated API requests.

Defending against XXE

- Most XXE attacks are both simple to exploit and simple to protect against. All modern XML parsers provide configuration options that allow the external entity to be disabled.
- More advanced XXE defense involves considering XML-like formats and XML-like parsers, such as SVG, PDF, RTF, etc., and evaluating the implementation of usage of those parsers in the same way you would a true XML parser to determine if any crossover functionality is present.

Defending against injection

- Injection attacks that target SQL databases can be stopped or reduced with proper SQL configuration and the proper generation of SQL queries (e.g., prepared statements).
- Injection attacks that target CLI interfaces are more difficult to detect and prevent against. When designing these tools, or im-

plementing one, best practices like the principle of least authority and separation of concerns should be strongly considered.

Defending against DoS

- DoS attacks originating from a single attacker can be mitigated by scanning regular expressions to detect backtracing problems, preventing user API calls from accessing functions that consume significant server resources, and adding rate limitations to these functions when required.
- DDoS attacks are more difficult to mitigate, but mitigations should start at the firewall and work their way up. Blackholing traffic is a potential solution, as is enlisting the help of a bandwidth management service that specializes in DDoS.

Defending against data and object attacks

- Attacks against data and objects typically occur due to either weak validation or weak data structure specification. By identifying common weaknesses that have been reported in other applications, data structures you can more effectively avoid these in your own applications.
- Some formats for storing data are inherently less secure than others. By developing an understanding of the pros and cons (from a security perspective) of all of the major data formats, you can better provide insight to your developers in regard to what data format should be used for what functionality.

Defending against client-side attacks

- Because client-side attacks occur within the browser, the first step for mitigating these attacks should always be looking to the browser in order to enable mitigations provided and maintained by the browser vendor. Tools like CSP provide a first line of defense in these cases.
- When first-line defenses are not sufficient, many client-side attacks can be mitigated by delving deep into JavaScript and the

DOM by making use of powerful functionalities like `Object.freeze()` or framebuster scripts.

Securing third-party dependencies

- Third-party dependencies are one of the security banes of modern web applications. Because of their rampant inclusion in first-party applications, combined with a mixed bag of security audits, third-party dependencies are a common cause of an application's demise.
- Third-party integrations should be integrated in a way that limits the integrations' permissions and scope to what is necessary. In addition, the integrations should be scanned and reviewed prior to integration. This includes looking into CVE databases to determine if any other researchers or organizations have reported vulnerabilities that affect the integration in question.

Mitigating business logic vulnerabilities

- Business logic vulnerabilities cannot be easily found with common tooling. They require a more hands-on approach to detection. Fortunately, comprehensive testing within the software development phase can eliminate some of the effort required.
- It's possible to build a model of your application's user workflows and automate the testing of various inputs, user flows, and commands. While this is typically not a comprehensive solution, it may help in identifying components of your codebase that are more vulnerable to business logic–related issues.

More to Learn

There is always more to learn. To become a web application security expert, you will need exposure to even more topics, technologies, and scenarios. You will have to refresh your knowledge on a regular basis, else you risk your knowledge becoming obsolete as the *modern* web applications you practiced on slowly transition to *legacy* web applications. This is simply a fact of life that is more apparent in our fast-moving industry. In this field of study, life-long learners excel the most.

While I did my best to cover as many important topics as possible in this book, it would be impossible to cover all relevant topics. The topics that ended up in this second edition were specifically chosen based on a few criteria:

- I wanted to make sure that each topic was applicable to a wide range of web applications because I wanted the book to be full of practical information you could digest and put to good use.
- Each topic had to be either at the recommended skill level or at a level that could be gained from studying previous chapters of the book. This means that the difficulty and knowledge required for each topic had to scale linearly with the previous knowledge presented. I couldn't skip around and expect the reader to find knowledge elsewhere; otherwise it would have become more of a glossary-style book instead of an immersive cover-to-cover read.
- Each topic in the book had to have some relation to the others for the book to flow easily from cover to cover. I found that in my own reading, few technical books and even fewer security books were organized carefully enough that I could just open one up and start learning where I left off without having to skip back and forth or consult a search engine.

It is my hope that the content within the pages of this book aid you in finding and resolving security flaws, enhancing your career, and piquing your technical interests. Beyond that, I hope it was an enjoyable and easy read—ideally one you will revisit and reference from time-to-time as needed.

Thank you for taking the time to read this book, and I wish you the best on your future security ventures.