Insecure by Default?
Authentication Services in Popular Web Frameworks

Hannah Li and David Evans, *University of Virginia*
⟨hannahli, evans⟩@virginia.edu

According to BuiltWith.com, 37% of Alexa's top 10K websites are built using one of the 7 most popular web frameworks (http://hotframeworks.com/). Hence, we hypothesize that the default authentication templates, tutorials, and documentation provided by these frameworks has a major impact on the security of many websites. This work studies how different design choices made by web frameworks impact the security of web applications built by typical developers using those frameworks. Our long-term goal is to understand the usability and performance trade-offs that lead frameworks to adopt insecure defaults, and develop alternatives that lead to better security without sacrificing the needs of easy initial development and deployment.

Our initial focus is on server-side frameworks that provide default authentication templates, documentations, or packages to help the developers get started. Of the 7 most popular frameworks, this excludes AngularJS (the second most popular) because it does not provide a native authentication package, reducing the percentage of top 10K sites covered to 30%.

Each of the three levels of aid—template, tutorial, documentation—from the framework expect increasing knowledge and expertise from the developer; for now, we do not consider third-party tutorials or templates. Our goal for this work is to identify factors that make a framework less secure than is possible following best known practices, and to understand why framework designers choose less secure options.

In addition to studying the frameworks and their documentation directly, we are building a tool called *AuthScan* to conduct automated large-scale scans of websites built with each framework to measure how well authentication is actually implemented for deployed web applications. Borrowing ideas from OpenWPM's structure (https://github.com/citp/OpenWPM/), AuthScan will use browser drivers and heuristics to automate the login and registration processes, which will check for authentication misuses in the web applications.

The rest of this abstract highlights preliminary findings from our manual analysis of popular frameworks.

Documentation and Templates. Although the settings added to default templates are obvious, directions written in tutorials and documentation are easier to miss. For example, while there is a page for Ruby on Rails on web security, it only provides descriptions of attacks and general steps to fix them, but it leaves it up to the developer to do things correctly. Additionally, documentation sometimes fails to provide clear cut directions to avoid some of the vulnerabilities. On the other hand, frameworks that provide login templates allow us to assess the security of web apps built using that template. A developer with limited security expertise is unlikely to change the default choices for password encryption and credential error messages. Those default choices are also influenced by trade-offs between security and functionality or convenience.

Security vs. Ease of Development. While most frameworks turn the HTTP-Only cookie flag on by default, the Secure flag has to be set manually. This is undesirable for security, but convenient from a development perspective because most websites are developed and tested using localhost. With the Secure flag turned on, cookies would not work because localhost does not have encrypted connection.

Security/Usability Tradeoffs. Frameworks also make decisions that may not be perceived as insecure, such as the choice of login error message and whether to include brute-force protection and login failure logging. For example, the code examples for Django and Meteor respectively give "inactive account" and "User not found" errors. While such specific error messages help the developer and user distinguish between non-existent user accounts and invalid passwords, accepted security practices encourage hiding this information for potential attackers. Other examples include brute-force protection and login failure logging, which are not widely adopted in the examined frameworks, although they have significant security benefits with minimal downsides. On the other hand, CRSF protections appear to be provided by all frameworks when applicable.

Conclusion. We expect most of the insecure aspects of current web frameworks are not due to ignorance or carelessness on the parts of their designers, but difficult tradeoffs between security, functionality, usability, and ease of development. We are optimistic, though, that better understanding of these issues will lead to alternative designs that can offer improved security within easy-to-use web frameworks.