Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery

Article *in* IEEE Security and Privacy Magazine · January 2020

DOI: 10.1109/MSEC.2019.2961649

5 authors, including:

Stefano Calzavara
Università Ca' Foscari Venezia
70 PUBLICATIONS **728** CITATIONS
SEE PROFILE

Mauro Conti
University of Padova
532 PUBLICATIONS **15,384** CITATIONS
SEE PROFILE

Riccardo Focardi
Università Ca' Foscari Venezia
163 PUBLICATIONS **3,432** CITATIONS
SEE PROFILE

Alvise Rabitti
Università Ca' Foscari Venezia
18 PUBLICATIONS **176** CITATIONS
SEE PROFILE

Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery

Stefano Calzavara*, Mauro Conti†, Riccardo Focardi*, Alvise Rabitti*, and Gabriele Tolomei‡
*Università Ca' Foscari Venezia, Italy
†University of Padua, Italy
‡Sapienza University of Rome, Italy

***Abstract*—In this article, we propose a methodology to leverage Machine Learning (ML) for the detection of web application vulnerabilities. Web applications are particularly challenging to analyse, due to their diversity and the widespread adoption of custom programming practices. ML is thus very helpful for web application security: it can take advantage of manually labeled data to bring the human understanding of the web application semantics into automated analysis tools. We use our methodology in the design of Mitch, the first ML solution for the black-box detection of Cross-Site Request Forgery (CSRF) vulnerabilities. Mitch allowed us to identify 35 new CSRFs on 20 major websites and 3 new CSRFs on production software.**

***Index Terms*—Machine learning, cross-site request forgery, web security.**

I. Web Vulnerability Detection

Web applications are the most common interface to security-sensitive data and functionality available nowadays. They are routinely used to file tax incomes, access the results of medical screenings, perform financial transactions, and share opinions with our circle of friends, just to mention a few popular use cases. On the downside, this means that web applications are appealing targets to malicious users (attackers) who are determined to force economic losses, unduly access confidential data or create embarrassment to their victims.

Securing web applications is well known to be hard [1]. There are several reasons for this, ranging from the heterogeneity and complexity of the web platform to the adoption of undisciplined scripting languages offering dubious security guarantees and not amenable for static analysis. In such a setting, *black-box* vulnerability detection methods are particularly popular [2], [3], [4]. As opposed to white-box techniques which require access to the web application source code, black-box methods operate at the level of HTTP traffic, i.e., HTTP requests and responses. Though this limited perspective might miss important insights, it has the key advantage of offering a *language-agnostic* vulnerability detection approach, which abstracts from the complexity of scripting languages and offers a uniform interface to the widest possible range of web applications. This sounds appealing, yet previous work showed that such an analysis is far from trivial [5], [6]. One of the main challenges there is how to expose to automated tools a critical ingredient of effective vulnerability detection, i.e., *an understanding of the web application semantics*.

A. Example: Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a well-known web attack that forces a user into submitting unwanted, attacker-controlled HTTP requests towards a vulnerable web application in which she is currently authenticated. The key concept of CSRF is that the malicious requests are routed to the web application through the user's browser, hence they might be indistinguishable from intended benign requests which were actually authorized by the user.

A typical CSRF attack works as follows (Figure 1):

1) Alice logs into an honest yet vulnerable web application, e.g., her preferred social network. Session authentication is implemented through a session cookie that is automatically attached by the browser to any subsequent request towards the web application;
2) Alice opens another tab and visits an unrelated website, e.g., a newspaper website, which returns a web page including malicious advertisement;
3) the malicious advertisement sends a *cross-site* request to the social network using HTML or JavaScript, e.g., asking to "like" a given political party. Since the request includes Alice's cookies, it is processed in her authentication context at the social network. This way, the malicious advertisement can force Alice into putting a "like" to the desired political party, which might skew the result of online surveys.

Notice that CSRF does not require the attacker to intercept or modify user's requests and responses: it suffices that the

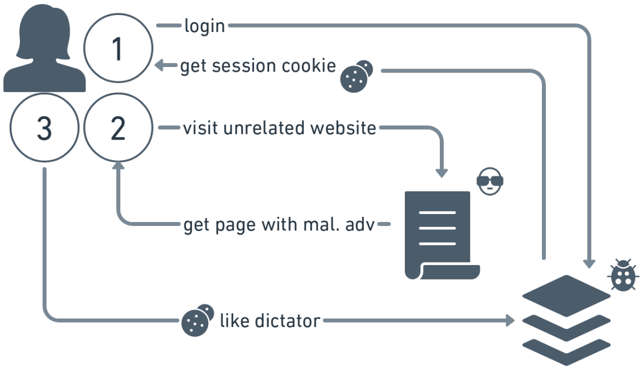

Fig. 1. Cross-site request forgery (example)

This article was supported by the project "Machine learning for web security", funded by the IRIDE program of Università Ca' Foscari Venezia.

victim visits the attacker's website, from which the attack is launched. Thus, CSRF vulnerabilities are exploitable by any malicious website on the Web.

B. Preventing CSRF

To prevent CSRF, web developers have to implement explicit protection mechanisms [7]. If adding extra user interaction does not affect usability too much, it is possible to force re-authentication or use one-time passwords / CAPTCHAs to prevent cross-site requests going through unnoticed. In many cases, however, automated prevention is preferred: the recently introduced SameSite cookie attribute can be used to prevent cookie attachment on cross-site requests, which solves the root cause of CSRF and is highly recommended for new web applications. Unfortunately, this defense is not yet widespread and existing web applications typically filter out cross-site request by using any of the following techniques:

1) checking the value of standard HTTP request headers such as `Referrer` and `Origin`, indicating the page originating the request;
2) checking the presence of custom HTTP request headers like `X-Requested-With`, which cannot be set from a cross-site position;
3) checking the presence of unpredictable anti-CSRF tokens, set by the server into sensitive forms.

A recent paper discusses the pros and cons of these different solutions [3]. However, all three options suffer from the same limitation: they require a careful and fine-grained placement of security checks. For example, tokens should be attached to all and only the security-sensitive HTTP requests, so as to ensure complete protection without harming the user experience. Using a token to protect a "like" button is useful to prevent the attack discussed above, yet having a token on the social network homepage is undesirable, because it might lead to rejecting legitimate cross-site requests, e.g., from clicks on the results of a search engine indexing the social network.

In the end, finding the "optimal" placement of anti-CSRF defenses is typically a daunting task for web developers. Modern web application development frameworks provide automated support for this, yet CSRF vulnerabilities are still routinely found even in top-ranked websites [2]. This motivates the need for effective CSRF detection tools. But how can we provide automated tool support for CSRF detection if we have no mechanized way to detect which HTTP requests are actually security-sensitive?

II. Machine Learning to the Rescue

The CSRF example in the previous section shows that it is useful to enrich vulnerability detection tools with *semantic* information so as to minimize their amount of false positives and false negatives. At the very least, one would desire a method to automatically classify HTTP requests as security-sensitive or not to restrict the analysis to the former. However, this is particularly challenging to do on the Web, since HTTP requests have a relatively weak syntactic structure and custom programming practices abound. For example, there are many different plausible ways to implement a "like" button for some content identified by the unique string `3aa5bf`, including:

1) a GET request to the page `like.php` with a single parameter `id = 3aa5bf`;
2) a GET request to the page `manage.php` with a parameter `id = 3aa5bf` and a parameter `action = like`;
3) a POST request to the page `manage.php` including a JSON object {`id: 3aa5bf, action: upvote`}.

All these requests look semantically similar to experienced security testers, yet they are syntactically different and it might be hard to identify all the most common ways to encode the same information in the wild.

A. Supervised Learning

Luckily, Machine Learning (ML) provides effective tools to automate classification tasks. A *classifier* can be seen as a function $f : \mathcal{X} \to \mathcal{Y}$ mapping any object from the *feature space* $\mathcal{X}$ into a corresponding *class* from $\mathcal{Y}$. The subfield of *supervised learning* studies effective techniques to automatically generate classifiers starting from a set of labelled data [8]. To fruitfully use supervised learning, one thus has to:

1) collect a set of objects of interest $\mathcal{O}$, for example HTTP requests sent to representative web applications;
2) define the set of classes $\mathcal{Y}$. For example, one could set $\mathcal{Y} = \{+1, -1\}$ to discriminate the security-sensitive requests (+1) from all the other ones (-1);
3) define the feature space $\mathcal{X}$ by manually identifying the salient aspects which look useful to assign the objects in $\mathcal{O}$ to their correct class in $\mathcal{Y}$. For example, one could leverage the request length, the request method or the presence of selected keywords in the request body;
4) build a *training set* $\mathcal{D}$ of pairs $(\vec{x}, y)$, where each $\vec{x}$ is the encoding in $\mathcal{X}$ of an object $o \in \mathcal{O}$ and y is its class.

Once this is done, supervised learning can automatically extract the best-performing classifier from a set of possible hypotheses $\mathcal{H}$ by estimating its performance on the training set $\mathcal{D}$. As long as one has enough manually curated data in $\mathcal{D}$, the performance of supervised learning can compete with or even outclass that of human experts [9], [10], [11].

B. Web Vulnerability Detection

At the end of the day, the methodology we put forward can be summarized as follows:

1) use supervised learning to automatically train a classifier which partitions selected web objects of interest, e.g., HTTP requests, HTTP responses or cookies, based on the web application semantics. For example, in the case of CSRF detection, the classifier would be used to identify security-sensitive HTTP requests;
2) for each possible class returned by the classifier, define a heuristic for vulnerability detection. Even trivial heuristics marking every object in a given class as non-vulnerable are plausible. For example, insensitive requests cannot be exploited for CSRF, hence they can be immediately marked as non-vulnerable;

3) use the classifier to choose the appropriate vulnerability detection heuristic to run on each web object of interest, e.g., as part of a browser extension.

We successfully leveraged this methodology in a number of research papers [12], [13], [14]. We report in the following on our most up-to-date study on CSRF detection [14].

III. MITCH: ML-BASED DETECTION OF CSRF

Mitch is the first tool for the black-box detection of CSRF vulnerabilities. Its design is based on the methodology presented in the previous section. Mitch is available online[1] as a browser extension for Mozilla Firefox. We refer to our recent research paper for full details [14].

A. Overview

Mitch assumes the possession of two test accounts (say, Alice and Bob) at the website where the security testing is to be performed. This is used to simulate a scenario where the attacker (Alice) inspects sensitive HTTP requests in her session to force the forgery of such requests in the browser of the victim (Bob). Having two test accounts is crucial for the precision of the tool because if the forged requests contain some information which is bound to Alice's session, then CSRF against Bob may not be possible. For example, if a website defends against CSRF through the use of anti-CSRF tokens, then Alice's requests will be rejected in Bob's session. The use of two test accounts for CSRF detection has already been advocated in previous work [2] and is part of traditional manual testing strategies.[2]

The architecture of Mitch is shown in Figure 2. After installing Mitch in her browser, the security tester first navigates the website as Alice: for every HTTP request detected as sensitive by the classifier, Mitch stores the content of the corresponding HTTP response. After completing the navigation, Mitch uses the collected sensitive HTTP requests to generate new HTML elements in the extension origin which allow for replaying them. The security tester then authenticates to the website as Bob and Mitch exploits the generated HTML to automatically replay the detected sensitive requests from a cross-site position, which simulates a CSRF attack. Finally, the responses collected for Alice and Bob are compared: if a response received by Bob "matches" the one received by Alice, it means that Alice was able to forge a valid request for Bob's session, hence the attack is considered successful and Mitch reports a potential CSRF vulnerability.

B. Challenges

The proposed CSRF detection heuristic is intuitive, yet there are several challenges to solve to make it work in practice. We provide a high-level view of such challenges and our proposed solutions below.

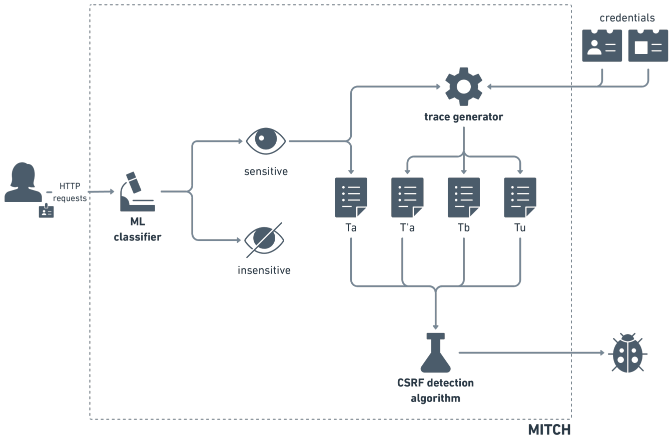

Fig. 2. Architecture of Mitch

1) Changes in HTTP Responses: Defining a suitable notion of "matching" HTTP responses for Alice's and Bob's sessions is generally hard, because HTTP responses may include dynamically generated elements, which might realistically differ even when the same idempotent operation is performed multiple times. Mitch thus builds on a notion of *dissimilar* HTTP responses. In general, the dissimilarity of HTTP responses is much easier to check than their similarity, e.g., due to the use of different status codes or content types to denote failures (for example, status codes 401 Unauthorized and 403 Forbidden are typical ways to denote unauthorized access). When Bob's response is dissimilar from Alice's response, it is likely that Alice's request failed in Bob's session, which might indicate the use of a CSRF protection mechanism.

2) Changes in Session State: Since the state of Alice and Bob at the website might be different, matching the response received by Bob against the one received by Alice might be an improper way to detect a CSRF vulnerability. For instance, Bob might not be able to perform a sensitive operation because it does not have access to the file `foo`, yet a CSRF attack would work if it targeted the file `bar`. When comparing the response received by Bob against the one received by Alice, Mitch does not immediately consider their dissimilarity as a definite evidence that the request of Bob had a different outcome than the one of Alice due to the use of a CSRF protection mechanism. Rather, since different outcomes might come from a difference in the state of Alice's and Bob's sessions, Mitch also replays the original request of Alice in a fresh Alice's session: if the new response received by Alice is dissimilar to the original one, it is likely that session-dependent information is required to process the request, which might indicate the adoption of an anti-CSRF token.

3) Classification Errors: Even a very accurate classifier might incorrectly mark an insensitive request as sensitive. In this case, there is no CSRF vulnerability and the presence of matching responses for Alice's and Bob's sessions should not raise an alarm. To detect potential false positives produced by the classifier, Mitch replays the original request of Alice without first authenticating to the website, i.e., outside any session: if the received response is dissimilar from the original

[1] https://github.com/alviser/mitch

[2] https://support.portswigger.net/customer/portal/articles/1965674-using-burp-to-test-for-cross-site-request-forgery-csrf-

one, then there is further evidence that the requested operation required an authenticated context to be performed, which confirms that there exists potential room for CSRF.

C. Machine Learning Classifier

The ML classifier used by Mitch was trained from a dataset of around $6,000$ HTTP requests from existing websites, collected and labeled by two human experts. The feature space $\mathcal{X}$ of the classifier has 49 dimensions, each one capturing a specific property of HTTP requests. Those can be organized into 3 categories: *Structural*, *Textual*, and *Functional*.

1) Structural: This category of features describes structural properties of an HTTP request. More precisely, we define the following set of numerical features:

- `numOfParams`: the total number of parameters;
- `numOfBools`: the number of request parameters bound to a boolean value;
- `numOfIds`: the number of request parameters bound to an *identifier*, i.e., a hexadecimal string, whose usage was empirically observed to be common in our dataset;
- `numOfBlobs`: the number of request parameters bound to a *blob*, i.e., any string which is not an identifier;
- `reqLen`: the total number of characters in the request, including parameter names and values.

While one might devise more sophisticated techniques to "type" request parameters, HTTP requests have a very weak structure and it is hard to come up with general yet accurate typing techniques for them.

2) Textual: This category of features captures textual characteristics of HTTP requests and is based on a small manually-curated vocabulary of keywords V that may occur in the request, resulting from a manual inspection of sensitive requests from a sample of real-world websites considered in our dataset. More specifically, we only consider binary features of the following forms:

- $word$`InPath`, where $word \in V$ means the presence of the string $word$ in the request path;
- $word$`InParams`, where $word \in V$ means the presence of the string $word$ in any parameter name of the request.

The vocabulary V includes the following 21 keywords, which have been selected as possible signals of sensitive requests, according to common sense and a preliminary inspection of the part of our dataset which is reserved for training: *create*, *add*, *set*, *delete*, *update*, *remove*, *friend*, *setting*, *password*, *token*, *change*, *action*, *pay*, *login*, *logout*, *post*, *comment*, *follow*, *subscribe*, *sign*, and *view*.

3) Functional: This category of features indicates the HTTP method associated to the request. We consider just the following two binary features:

- `isGET`: the HTTP request method is GET;
- `isPOST`: the HTTP request method is POST.

There are no additional alternatives, because our dataset only includes GET and POST requests. All the other requests can be easily labelled as sensitive or not just based on their method, e.g., OPTIONS requests are always insensitive.

IV. Experimental Evaluation

In this section, we evaluate the effectiveness of Mitch in detecting CSRF vulnerabilities. In particular, we show that the number of false positives and false negatives produced by Mitch is remarkably low and amenable for practical use.

A. False Positives and False Negatives

Mitch produces a false positive when it returns a candidate CSRF that cannot be actually exploited. This is something relatively easy to detect by manual testing, though this process is tedious and time-consuming. In general, it is not possible to reliably identify when Mitch produces a false negative, because this would require to know all the CSRF vulnerabilities on the tested websites. To estimate this important aspect, we keep track of all the sensitive requests returned by the ML classifier embedded into Mitch and we focus our manual testing on those cases. This is a reasonable choice to make the analysis tractable, because we first showed that the classifier performs well using standard validity measures.

B. Assessment on Existing Websites

To test how effective is Mitch on existing websites, we sampled 20 websites from the Alexa Top 10k ranking. We only considered websites with single sign-on access via a major social network website, so we could leverage just two existing social accounts to perform our security testing.

Overall, Mitch found 191 sensitive requests and reported 47 potential CSRF vulnerabilities: we were able to immediately exploit 35 of them, exposing major security issues in a few cases. We estimated only 7 false negatives in total, which means that our heuristics are accurate enough to capture most of the vulnerabilities. The full breakdown of the individual websites is shown in Table I and commented below.

Many of the attacks we found targeted the social functionalities of the websites we tested, like casting votes on public

Website	*Sensitive Requests*	*Detected CSRFs*	*fp*	*fn*
9gag.com	10	3	1	0
ask.fm	16	0	0	0
askubuntu.com	16	0	0	0
bombas.com	2	1	0	1
brilio.net	2	1	0	1
eprice.it	11	3	0	3
flixbus.com	4	1	1	0
funnyjunk.com	17	8	2	2
gsmarena.com	3	3	0	0
imdb.com	10	0	0	0
imgur.com	12	3	3	0
indeed.com	8	4	0	0
instructables.com	11	4	0	0
mocospace.com	7	5	2	0
pornhub.com	13	2	1	0
smokecartel.com	5	2	0	0
starnow.com.au	8	4	0	0
tomshardware.com	13	1	1	0
wish.com	11	0	0	0
yelp.com	12	2	1	0
TOTAL	191	47	12	7

TABLE I
CSRF DETECTION ON EXISTING WEBSITES

contents, adding or removing items from favorite lists, and posting comments under the identity of the victim. Most of these attacks may thus affect recommender systems, lead to social embarrassment, and compromise user reputation at scale. Worse, we were also able to find a number of nasty attacks which seriously compromised the website functionality; we responsibly disclosed all the vulnerabilities to the respective website owners. We discuss a few interesting cases below.

1) Bombas: Bombas is an e-commerce website selling socks. It provides a functionality to store a list of shipping addresses to simplify purchases, so that shipping details do not need to be entered for each transaction. The form used to store a new shipping address is vulnerable to CSRF, so an attacker can force any address into the victim's account to hijack deliveries. Notice that the latest added address is the one which is used by default, which makes the attack even worse in terms of practical impact.

Remarkably, Bombas is a customer of Shopify, which is a major e-commerce platform, so this attack may also affect many other websites. We reported the issue to Shopify, which acknowledged the attack and is working on a fix, but marked our report as duplicate due to the existence of a previous independent disclosure.

2) Indeed: Indeed is one of the biggest websites hosting job offers. Registered users can send their CVs and apply to different open positions in the world. We found three CSRF vulnerabilities which give an attacker the ability of fully managing the job offers associated to the account, including the possibility of storing new offers and archiving existing ones. Indeed also suffers from a CSRF vulnerability on the form used to set user preferences, which can be used to severely affect the visibility of job offers. An attacker can exploit this vulnerability to hide job offers, for instance by restricting the search radius and changing the desired publication date for displayed offers.

We find these vulnerabilities particularly interesting, because Indeed is making wide use of anti-CSRF tokens and all the vulnerable forms have their own token. However, it seems that not all the tokens are correctly checked by the website, which may suggest a manual, error-prone placement of the tokens. More generally, this shows that checking the presence of anti-CSRF tokens is not sufficient to say that a website is protected against CSRF, and that the actual website behavior should be tested instead. The security team of Indeed acknowledged the issue and rewarded us $100 for the finding.

3) Starnow: Starnow is an Australian website designed to discover new talents, such as singers and actors. Users who are interested into pursuing an artistic career can register to the website to get access to a number of auditions and job interviews. The first two CSRFs we found allow an attacker to arbitrarily manipulate the watchlists of authenticated users, thus compromising a functionality offered by the website.

There are however two much worse attacks. A CSRF vulnerability affects the form used to store the phone number associated to user profiles: this can be used for scams or to disrupt the functionality of the website, e.g., by making impossible to contact the victim for an audition. It is interesting that the request used to set the phone number contains an anti-CSRF token, which however is not checked by the website: this confirms that this kind of mistakes is not confined to Indeed, but is apparently more widespread.

Web application	*Sensitive Requests*	*Detected CSRFs*	*fp*	*fn*
Oxid e-shop 4.9.8	21	4	1	0
Prestashop 1.6.1.2	12	1	1	0
SM Forums 2.0.12	9	0	0	0
TOTAL	42	5	2	0

TABLE II
CSRF DETECTION ON PRODUCTION SOFTWARE

The last CSRF vulnerability is definitely the most severe one, because it affects the form used to set the email address of user profiles. By exploiting this vulnerability, the attacker can set the victim's email address to her own one and then use the password reset functionality of Starnow to get a fresh password for the victim in her inbox, thus taking possession of the victim's account.

C. Assessment on Production Software

As a second set of experiments, we decided to run Mitch on the testbed of open-source web applications used to evaluate Deemon, a state-of-the-art automated detection tool for CSRF vulnerabilities [15]. Notice that, since Deemon only works on PHP applications whose source code is available for dynamic analysis, we could not test it on the closed-source websites from our first set of experiments. Out of the 10 applications considered in the original testbed, we were only able to find 3 applications at the same version: Oxid e-shop, Prestashop and Simple Machine Forums. No CSRF vulnerability was detected by Deemon on these applications, according to the experimental evaluation in [15]. The results of the analysis performed by Mitch on the applications in their default configuration are shown in Table II.

Mitch was extremely effective on the tested applications, because it reported only 2 false positives and it was able to catch 3 CSRF vulnerabilities on Oxid e-shop which were not reported by Deemon [15]. These vulnerabilities allow an attacker to corrupt the integrity of the shopping cart, force the use of vouchers and change the preferred payment method. Remarkably, all the corresponding functionalities are supposed to be protected by an anti-CSRF token, which however is not checked by the Oxid back-end. We reported the issues to the Oxid security team, who acknowledged the problem and worked on a fix.

V. FREEWARE AND OPEN-SOURCE SOFTWARE

Penetration testers have been using a range of different tools to detect CSRF vulnerabilities in web applications. Based on an extensive research on blogs, forums and resources for security practitioners, including the OWASP Testing Guide, we classified existing tools in the following categories:

1) *intercepting proxies*: these tools allow penetration testers to intercept and modify arbitrary HTTP traffic, which can be used for an essentially manual detection of web vulnerabilities, including CSRF. Popular tools in this category are Burp, ZAP, and WebScarab;

2) *exploit generators*: these tools simplify the generation of proof of concepts for attack finding, based on human guidance on the set of HTTP requests which need to be tested for CSRF. Examples tools in this category include CSRFTester and pinata-csrf-tool;
3) *web application scanners*: these tools automatically detect a range of web application vulnerabilities, including CSRF, based on different heuristics. Scanners supporting modules for CSRF are Arachni, Skipfish, and w3af.

Our work improves on 1) and 2) by providing effective automated techniques for the detection and the exploitation of sensitive HTTP requests, as opposed to manual investigation and testing. The most important advances over 3) are instead the use of machine learning for sensitive request detection, a more sophisticated CSRF detection algorithm and a systematic evaluation of the performance of our detection tool, based on the analysis of false positives and false negatives produced on real web applications. Remarkably, we noticed important design limitations in the opensource tools we analyzed, which significantly downgrade their accuracy.

For example, Arachni only detects CSRF vulnerabilities on forms requiring an authenticated context, hence it does not capture CSRF attempts via links or AJAX. The rationale behind this choice is likely the complexity of detecting sensitive HTTP requests, which forced the developers of Arachni to limit their tool to HTML elements which are potentially dangerous, yet easy to catch syntactically (forms). It is instructive that w3af suffers from a somewhat opposite design choice: since any request which includes cookies and parameters is deemed as potentially sensitive by w3af, the tool is affected by a very large number of false positives, which led to the opening of an issue on GitHub where a major redesign of the tool is advocated[3]. Other issues we found are related to the choice of deeming secure any HTTP request which includes an anti-CSRF token, while we observed several cases where tokens are not checked at the web application back-end, and to the use of just a single authenticated session, which loses precision when user-dependent secrets happen to thwart CSRF attempts. In the end, preliminary tests with existing web application scanners on simple examples returned a high number of false positives and false negatives, which is in line with the findings of previous research work which showed the ineffectiveness of such scanners for CSRF detection [5], [6].

VI. Conclusion

Web applications are particularly challenging to analyse, due to their diversity and the widespread adoption of custom programming practices. ML is thus very helpful in the web setting, because it can take advantage of manually labeled data to expose the human understanding of the web application semantics to automated analysis tools. We validated this claim by designing Mitch, the first ML solution for the black-box detection of CSRF vulnerabilities, and by experimentally assessing its effectiveness. We hope other researchers might take advantage of our methodology for the detection of other classes of web application vulnerabilities.

[3] https://github.com/andresriancho/w3af/issues/120

References

[1] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. Surviving the web: A journey into web session security. *ACM Comput. Surv.*, 50(1):13:1–13:34, 2017.

[2] Avinash Sudhodanan, Roberto Carbone, Luca Compagna, Nicolas Dolgin, Alessandro Armando, and Umberto Morelli. Large-scale analysis & detection of authentication cross-site request forgeries. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 350–365, 2017.

[3] Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi. Testing for integrity flaws in web sessions. In *Computer Security - 24rd European Symposium on Research in Computer Security, ESORICS 2019, Luxembourg, Luxembourg, September 23-27, 2019*, pages 606–624, 2019.

[4] OWASP. OWASP Testing Guide. https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents, 2016.

[5] Jason Bau, Elie Bursztein, Divij Gupta, and John C. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, pages 332–345, 2010.

[6] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*, pages 111–131, 2010.

[7] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 75–88, 2008.

[8] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[9] Michael W. Kattan, Dennis A. Adams, and Michael S. Parks. A comparison of machine learning with human judgment. *Journal of Management Information Systems*, 9(4):37–57, March 1993.

[10] D. A. Ferrucci. Introduction to "This is Watson". *IBM Journal of Research and Development*, 56(3):235–249, May 2012.

[11] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[12] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[13] Stefano Calzavara, Gabriele Tolomei, Andrea Casini, Michele Bugliesi, and Salvatore Orlando. A supervised learning approach to protect client authentication on the web. *TWEB*, 9(3):15:1–15:30, 2015.

[14] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 528–543, 2019.

[15] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1757–1771, 2017.

Stefano Calzavara is a tenure-track assistant professor at Università Ca' Foscari Venezia, Italy. He received a PhD in Computer Science at Università Ca' Foscari Venezia, Italy, in 2013. His main research interests are formal methods and web security. Contact him at calzavara@dais.unive.it.

Mauro Conti is a full professor at University of Padua, Italy. He received a PhD in Computer Science at Sapienza University of Rome, Italy, in 2009. His main research interestes are computer security and privacy. Contact him at conti@math.unipd.it.

Riccardo Focardi is a full professor at Università Ca' Foscari Venezia, Italy. He received a PhD in Computer Science at University of Bologna, Italy, in 1999. His main research interests are computer security and formal methods. Contact him at focardi@unive.it.

Alvise Rabitti is a security officer at Università Ca' Foscari Venezia, Italy. He received a bachelor degree in Computer Science from Università Ca' Foscari Venezia, Italy, in 2013. His main research interests are web security and privacy. Contact him at alvise.rabitti@unive.it.

Gabriele Tolomei is an associate professor at Sapienza University of Rome, Italy. He received a PhD in Computer Science at Università Ca' Foscari Venezia, Italy, in 2011. His main research interests are machine learning and web search. Contact him at tolomei@di.uniroma1.it.