Automated Security Testing for Applications Integrating Third-Party Services

Project Description

Modern applications have become increasingly complex in both function and construction. Commerce websites use inferred user preferences to show relevant merchandise, banking websites implement complex transaction protocols, social networks need to safeguard sensitive user information, and mobile applications incorporate authentication, sharing, and payment mechanisms. Third-party services have become a common way to implement these functionalities, effectively making most applications today large mashups of both code and services from numerous parties. Widespread integration of third-party services into web and mobile applications raises a critical problem: *how to ensure desired security and privacy properties for programs integrating third-party services*.

Throughout this proposal we define a *third-party service* as a module or code snippet that is provided by a party other than the application developer and which communicates with its own back end server to which the application developer has limited access. Although our preliminary focus has been on web applications (that is, applications where the client side runs in a web browser), our target also includes mobile applications (applications where the client side runs on a mobile device platform such as Android) which face many of the same issues.

We focus on third-party services that provide critical functionalities. Some examples include authentication, authorization, file sharing and payment services. In the authentication context, a typical desired security property could be to ensure the identity consistency of the authenticated user, whereas in payment service context the security property of interest may be to ensure the information on the invoice and receipt be consistent with the actual monetary transaction. We assume the service provider is benign and trusted and the threats are external; if not, the integrator should not be relying upon it for a security-critical function.

There is also another important type of third-party services that do not provide critical functionalities — a typical example would be web analytics and advertising scripts. Although such (untrusted) services also bring security and privacy risks to users of applications that embed these services, they are quite different from the ones we focus on in this proposal. In this work, we do not consider services where the primary concern is malicious behavior from the service provider, but focus on the scenario where the integrator and service provider share the common goal of constructing a secure application.

Overview of Proposed Work

Developing stand-alone programs that satisfy desired security properties is a long-standing hard problem, although one for which the research community is making considerable progress. Creating secure integrated applications poses additional challenges, however, due to limited visibility of important system components. The server-side code for external services is typically unavailable to the integrator, and the server-side code for the application is unavailable to an outside tester. Although static analysis [4, 3, 25, 14] and dynamic analysis [10, 11, 9, 35, 13] techniques have many advantages over testing in improving security of stand-alone programs, we focus on dynamic testing approaches because we target systems that include critical components which are not available for program analysis.

Systematic Vulnerability Discovery (Section 1). We propose to develop a systematic method to discover potential pitfalls in integrating third-party services that provide critical security functions. Our method will identify the assumptions that must be ensured to use the service security, and provide the basis for automated tests that can reveal when an attacker can violate those assumptions. Our modeling approach still requires some manual effort, but prescribes a systematic and iterative process that is largely automated. Ideally, the explication process would become part of the development practice followed by service providers to fix problems and improve documentation before they release their SDKs. Service providers can use available source code implementations to automate more aspects of the explication process. Key to our approach

is the use of formal techniques that can explore the space of all possible reasonable applications, which exhaustively examining sequences of actions an attacker may take to exploit those applications.

Automated Enrollment (Section 2). The vulnerabilities we are targeting involve transactional aspects of web and mobile applications, including those that involve authenticated users. Hence, large-scale automated scanning requires being able to automatically register accounts and log into the applications. Applications have traditionally been designed to discourage automated account creation, but increasing adoption of single sign-on services provides a new opportunity for automating registrations. Our initial prototype has shown promising results — it can fully automatically create and log into test accounts for approximately 80% of websites that implement single sign-on using Facebook Connect. We propose work to further improve the automation success rate and to support additional identity providers and protocols, as well as to develop a tool to provide automated enrollment for testing Android applications.

Scanning Service Integrations (Section 3). We propose to build vulnerability scanners to perform deep scanning for applications integrating third-party services. We describe on our preliminary experiments with a system for testing web applications for vulnerabilities in their integration of Facebook's SSO service. We report on our results scanning the top 20,000 websites which that out of the 1700 that embed Facebook SSO, 345 are vulnerable to serious impersonation or credential leaking attacks. We propose work to extend our prototype deep scanning tool cover more services and mobile applications (Section 3.2.1).

In addition to performing large scale scans, we envision the vulnerability scanner being deployed by an application distribution center (e.g., Google Play scan Android apps for vulnerabilities) or service provider (e.g., Facebook would scan applications that use Facebook SSO for vulnerabilities). Service providers have a strong motivation to ensure security properties of applications that use its service, and could use the scanner to identify applications that can compromise their users.

The other deployment opportunity for the vulnerability scanner is as a stand-alone service that developers may use to check their implementations before they are launched (we have released an example of such a service based on our preliminary work with Facebook SSO, `ssoscan.org`). This usage scenario opens the possibility of using humans to provide guidance to the testing tools by using a browser extension to train the scanner to perform interactions cannot be fully automated (Section 3.2.2).

1 Explicating SDKs

Despite efforts by service providers to design easy-to-use software development kits (SDKs), developers often make integration mistakes that result in serious security vulnerabilities. Most mistakes are due to failures to understand and ensure assumptions needed to correctly use the SDKs. Previous works identified likely vulnerabilities using ad-hoc approaches [36, 38, 42, 41]. Our goal is to systematize and automate the process of identifying pitfalls in integrating SDKs by explicating the assumptions needed to use them securely.

Our preliminary work developed a systematic approach to explicate SDKs for authentication (Section 1.1), and confirmed that many deployed web applications fail to ensure these assumptions with serious consequences (an example is presented in Section 1.1.2). We propose to build on this work to develop a more general explication approach that works for a variety of third-party services including authentication, payment, and file sharing services (Section 1.2). We also propose to improve the explication process by automating more aspects of the modeling process using automated translation when source code is available, and automated behavior probing for modules available only as black boxes (Section 1.2).

1.1 Preliminary Work: Explicating Single Sign-On SDKs

As preliminary work, we studied two SSO services that are widely deployed in popular web and mobile applications and critical for security: Facebook SSO and Windows Live Connect. Our experience from this work provides the foundation, as well as the motivation, for the proposed work on improving the explication process, as well as the basis for several of the vulnerabilities we check for in our preliminary scanning experiments (Section 3.1). Next, we briefly describe the modeling process (Section 1.1.1) and summarize our experiments with SSO services (Section 1.1.2). See Wang et al. [43] for more details.

1.1.1 Modeling SDKs and Services

Figure 1 depicts the iterative process we use to explicate an SDK. It involves building a model based on the system behavior and checking it against desired security properties using a model checker or theorem prover. During the checking process, counter-examples reported by the model checker either lead us to refine the model or reveal required assumptions or bugs in the SDK. This approach gives stronger completeness properties than ad hoc approaches, and provides confidence that the assumptions found are complete with respect to the model.

Related work. Our approach is inspired by prior works that model programs using counter-example guided abstraction refinement including SLAM [4] and Microsoft's static driver verifier [3], as well as verification and inference techniques including interface verification [37, 16, 7], invariant extraction [23, 21, 22, 26, 49, 50] and specification mining [44, 30]. Unlike those projects which work on program code or execution traces, our target is a multi-faceted system that includes external components which cannot be examined directly. For most third-party services, we only have access to a provided interface and natural language documentation. Thus, building the models involves inference from experiments and human interpretation of documentation. Because of this, many previous techniques such as cannot be applied directly (although as we discuss in Section 1.2, we propose to develop automated inference techniques to reduce human effort required in the modeling process).

Defining security properties. A precise understanding of the security properties that an integrated service should provide is essential for explicating the SDK. The formal security properties that determine which assertions should be added to the model since those are the assertions needed to guarantee the desired security properties. Although desired security properties vary widely across different services, we have identified several common principles that should hold across service APIs: 1) *verification* — a secret or signature needs to be verified before it is trusted; 2) *secrecy* — user credentials or application secrets must not be leaked through any API calls; 3) *consistency* — bindings of user data must be consistent across transactions; and 4) *complete mediation* — every access to a protected resource must be check for appropriate

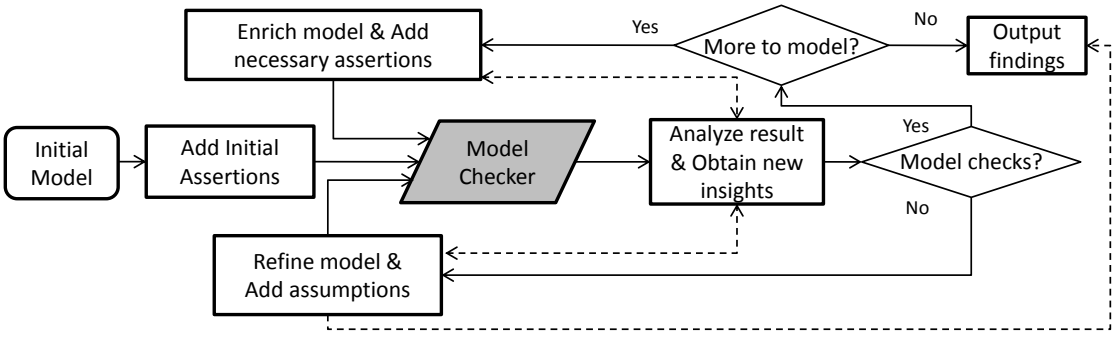

Figure 1: Iterative process to uncover security-critical assumptions.

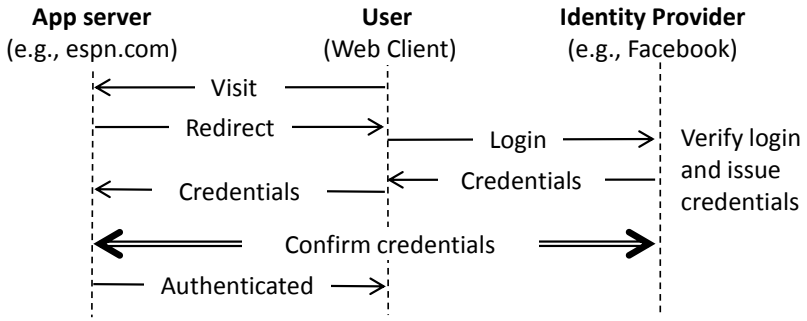

Figure 2: Single-Sign On Process

authorization. We demonstrate how this aids precise understanding of desired security properties for single sign-on services in Section 1.1.2.

Developing the model. We begin building our model with an initial set of APIs that are relevant to checking the security properties. These APIs are either explicitly documented in the developer guide or appear in sample applications. The initial function models can just be placeholders denoting (incorrectly for now) that the modeled API call does nothing. Then, we refine the model based on results from the model checker.[1]

At an early stage, counter-examples found by the model checker are usually due to the inaccuracy of the model. Based on a manual investigation of the counter-example path, we refine the model by using information either derived from the source code (when available), or the documentation and probing experiments. In later stages, counter-examples are more likely to be legitimate threats. To confirm a potential threat, we map the execution path to a real-world scenario and try to create an end-to-end exploit to verify the vulnerability. This reveals either a bug in the SDK code itself, or a missing implicit assumption needed to assure security. We document the assumption and insert it into in our model (using an *assume*), so that the model checker will consider the vulnerability fixed and continue to look for other counter-example paths in the next run. These uncovered assumptions are the key result of our explicating process. They lead directly to the security tests which check if applications allow those assumptions to be violated.

1.1.2 Experiments with Single Sign-On Services

We applied the explication process to two popular single sign-on SDKs: Facebook's PHP SDK and Windows Live SDK. SSO services allow users to log into an application using popular social network accounts such as Facebook or Twitter and connect their account on the new site to an established Internet identity, instead of establishing a new site-specific account.

A typical SSO process involves three parties and several steps, as depicted in Figure 2. A user first visits a web application and elects to use SSO to login. She is then redirected to the identity provider's SSO entry point (e.g., Facebook's server). After she logs into the identity provider, her SSO credentials are issued to the application server. The application server confirms the identity by either decoding the credential or further communicating with the identity provider's server, and finally authenticates the client as Alice. Many SSO services use the standardized OAuth 2.0 protocol[27].

Defining security properties. For SSO services, we define three important security properties following the guidelines we described in Section 1.1.1: 1) *authentication* — the application must verify the identity and signature of submitted credentials upon consumption to ensure protection against impersonation attacks; 2)

[1]In our preliminary work, we have used both Corral [29] as the model checker and the Boogie theorem prover [5], but our design is not tied to particular tools.

authorization — OAuth tokens bearing user permission must stay within the session, ensuring no unsolicited permission leakage; and 3) *association* — the user identity of the application must match the identity of her associated OAuth tokens, ensuring consistencies between OAuth credentials and session identities.

Modeling participating parties. Our goal is to establish that any application written following states assumptions will ensure the security properties no matter what the adversary and normal end users do. This means that in addition to modeling the APIs provided by the identity providers as we described in Section 1.1, we also need to model the behavior of three additional this: the application, the adversary, and the end user. These three parties are all abstract modules — they do not have a concrete implementation and the model should capture all reasonable behaviors of possible applications and end users, and all possible behaviors of an adversary. The application's behavior will be subject to the developer guide, but the adversary may call any exposed APIs in any order with any arguments. We use a knowledge pool to represent knowledge acquired by the adversary, and the attacker may use the knowledge pool to select arguments for API calls. Initially the knowledge pool contains publicly available information such as the benign application's unique application ID. As the adversary proceeds to call those APIs it may discover new values, which are added to the knowledge pool. If some sequence of adversary, user, and application actions leads to the adversary obtaining information that allows it to violate one of the desired security properties, this will manifest as an assertion failure and an analysis of the resulting counter-example will reveal either a missing security-critical assumption (which will be documented and added to the application model) or a bug in the SDK.

Findings. Adding assertions based on the above security properties, the explication process led us to find 3 bugs in the Facebook SDK[2] and identify 9 undocumented security-critical assumptions.

An example of one of the more common assumptions missed by many apps is that the identity provider assumes the developer will never use access token for authentication purposes. An access token should only be used to authorize the token bearer to perform actions on behalf of the user, such as posting feeds or accessing friend information of that user. When an application misuses such token to authenticate users, it presents an impersonation attack opportunity for any adversary that possesses an access token from the victim user, even if the token was issues for some other application. Such an access token is also easy to obtain, as the adversary may convince the user to use a rogue application or an access token from another vulnerable application.

The explication process guided us to uncover this missing assumption when the model checker found an execution path violating the assertion that the malicious party's authentication cookie cannot have victim's identity (this is the authentication property defined earlier). The counter-example path included two functions that that used access tokens inconsistently, allowing us to identify the missing assumption as well as to quickly pinpoint the key steps needed to exploit the vulnerability.

1.2 Proposed Work: Improving Explication Process

Our preliminary work provides encouraging support for the theses that a systematic process can be used to uncover important security assumptions, and that many web applications suffer from serious security vulnerabilities due to failures to ensure those assumptions. The most important limitation of the explication process is that it involves substantial human effort which also presents opportunities for human error. While we believe picking the relevant modules to model and mapping counterexamples to real world threats always requires domain expertise and is hard to automate, we believe other parts of the process can be automated. To enable more comprehensive analysis, and reduce the effort required to explicate a new SDK, we propose to develop methods to further systematize and automate the model-building process. We discuss two directions next: the first applies where source code is available, the second only assumes black-box access. Section 1.3 describes our plans to evaluate these improvements by applying our approach

[2]We reported the bugs to Facebook, who confirmed them, and awarded bug bounties for each of the bugs.

to various other services and platforms. We are particularly concerned with making the effort required to test a revision to an SDK or service small enough that the process can be repeated for each update release.

Automated model extraction. We propose to leverage program analysis techniques to construct an automatic model extractor for modules for which source code is available. This includes at a minimum all the code provided in the SDK that is used by developers. For some services may also include at least some parts of the external server, and all of that code when the explication is done by the service provider. Automatic translation an implementation language to model checking language might not seem to be an interesting research problem (indeed, some model checkers are able to work on implementation languages like C directly), but a strict semantics-preserving translation is not desirable for the model. Model checkers are limited by the space explosion problem and cannot check complex programs in a reasonable amount of time or memory. In our SSO study, we found it necessary to manually omit irrelevant variables and code (e.g., updating the GUI interface or logging messages) and simplify data structures (e.g., replacing array variables and complex data structures with single variables or tuples in ways that preserved the properties relevant for security) to allow the model checker to finish within several hours.

Given a list of variables of interest derived from the security properties, the model extractor should be able to ignore irrelevant variables, simplify data structures, and translate the remaining logic into the modeling language. To filter out irrelevant variables, we plan to employ taint tracking and program slicing techniques. The extractor should also abstract complicated data structures into simpler data structures such as fixed tuples that still capture the properties relevant for security.

Automated behavior probing. For black box modules and APIs, our preliminary work used manual probes and observations to guess their behavior and build a model. This is time-consuming and error-prone. We propose to develop techniques to partially automate this process, especially for straightforward client-server communication APIs with clearly visible effects in the response. The proposed observer can be implemented in a proxy server such as Fiddler, requesting target APIs with different parameters, and generalizing a model by analyzing those responses.

One challenge is how to supply meaningful inputs to the request and to extract meaningful information from the responses. Human effort may still be required to identify input/output types for the observer, although we expect for many client-server communication APIs the space of possible input types is small enough that it can be searched heuristically. A skeleton of the observed API may need to be created manually and the observer can fill in its body based on what it learns from the probing.

Another limitation of this approach is that our probe has no way of observing server state. Although server state changes due to API calls may be visible to the observer through subsequent calls, such state changes would be hard to determine without understanding what the API does. We will evaluate how serious this limitation is when applying the explication process to more services (Section 1.3), and explore techniques for modeling relevant server state based on request responses.

1.3 Proposed Work: Explicating Other Services

Our preliminary work demonstrates that the explication approach can indeed uncover implicit assumptions that are often missed by developers, and in Section 3.1 we provide evidence that such failures commonly lead to serious security vulnerabilities in deployed applications. To evaluate the improvements we propose in the previous subsection, as well as to expand the scope of applications and services we can check, we will apply the improved explication process to several additional services including payment and file sharing APIs described below. One important aspect of explicating new services is defining a precise set of security properties for each service, following the general guidelines introduced in Section 1.1). In addition, by explicating a variety of SDKs, we hope to learn more about how SDK design decisions can impact the risk

of dangerous implicit assumptions.

Target services. Payment services such as Paypal and Braintree performs payment processing for online vendors. As a user attempts to pay the vendor, she is redirected to the payment service where she pays and receives a receipt, which is then forwarded back to the vendor. To verify the authenticity of the receipt, the vendor can check the signature or send a request to the payment service depending on different implementations. We propose to check for several security properties regarding payment services: 1) *identity consistency* — the buyer's identity in the merchant application must match the payer's identity in the payment service; 2) *amount consistency* — the amount paid to the merchant must equal to the marked price; 3) *payee verification* — the merchant must verify the payee in the payment service is the merchant themselves; 4) *token verification* — the merchant must verify the correctness and validity of the payment token or signature if applicable.

File sharing services such as Dropbox and Box enables integrators to easily access user's files stored in their services. Common features using these services include sharing files with the integrators' application (via downloading from file sharing services) and saving user data to the cloud(via uploading to file sharing services). Key security properties include: 1) *secrecy* — the file owner's resource and access token must be stored securely and not be leaked to unauthorized parties; 2) *verification* — the application must verify the authenticity of an user action related to file access to protect against CSRF attacks; and 3) *consistency* — if the integrator stores the user's file sharing service credentials in a back end database, the credentials' identity must match their associated user's identity.

Probing challenges. Automatic behavior probing for these types of services poses additional challenges. Learning behavior of SSO APIs only requires sending and receiving simple HTTP GET and POST requests/responses, but this may not be the case for other services. For example, to observe certain file sharing APIs, the probing system may trigger uploading and downloading of files, and it needs to understand file content and directory structures to learn the output of the call. For online payment APIs, the security protections may be tighter which will likely limit our capabilities to create fake accounts and try (potentially) illegal transaction-related requests in the behavior probing process. We will develop heuristics to support semi-automatic behavior in these scenarios (which could also benefit from the human-guided testing tools we propose in Section 3.2).

2 Automating Enrollment

Recently, several large scale studies have examined security and privacy risks of integrated applications including private data exfiltration, browser fingerprinting, malware detection, and malicious advertising [1, 33, 40, 31, 32, 20, 53]. Such studies provide important insights into the state of security today for users, developers, service providers, and security researchers. A common limitation in these studies is that the analysis is always limited to anonymous sessions or small-scale authenticated sessions which are done manually. However, most critical vulnerabilities, including ones that involve failures to ensure the assumptions we uncover through the explicating process, concern sensitive transactions that depend on authenticated users interacting with the application in complex ways. To detect these vulnerabilities, we propose to develop techniques that enable automated deep scanning of complex web and mobile applications. These techniques involve *automating the login process* (the focus of this section) and *developing oracles that can detect transactional vulnerabilities* (the focus of Section 3).

2.1 Opportunity

To deter spammers, most websites are designed to make automating enrollment difficult, typically by requiring new users to solve a CAPTCHA.[3] However, the growing popularity of single sign-on services presents a new opportunity since applications do not typically require the user to complete CAPTCHAs to enroll using SSO. This is reasonable since the user has already proved to the application that she owns a valid account the identity provider (e.g., Facebook), and the application trusts the identity provider to make it difficult to establish those accounts. Our preliminary study (Section 2.2) confirms this assumption: less than 4% of applications we tested require the user to solve a CAPTCHA to enroll a new account and login using single sign-on.

Since applications provide different interfaces to their SSO login, automating enrollment requires a goal-directed exploration along with techniques for verifying if the process is progressing to the goal of enrolling and logging into an authenticated account in the application. Since each user interaction with the application is likely to trigger round trip traffic and a non-trivial delay to get the response, our primary focus is to develop useful heuristics to quickly prune the search space and improve the likelihood of efficiently finding the necessary sequence of actions.

Several tools have been created for automating GUI testing based on manually-generated scripts, including Selenium [18], TestingBot [39], and BugBuster [8]. These tools simply replay manually-created site-specific testing scripts.[4] Our goal is to develop tools that can automatically be effective on most websites and mobile applications. Our approach builds on recent works on automated GUI element triggering to explore application execution space on Android system [52, 34], Windows applications [45], and web applications [6, 28]. A common goal of these works is to explore an application's execution space to discover buggy, abnormal or malicious behavior. By contrast, our goal is to drive an application though a specific SSO process.

Next, we summarize our preliminary work implementing an automated enroller for web applications using Facebook SSO. Section 2.3 discusses opportunities to improve the automation success rate and expand support to additional SSO identity providers and platforms. Being able to automatically create an authenticated session opens up new opportunities for automated security testing, which we discuss in Section 3.

2.2 Preliminary Work

As preliminary work, we built an automatic account registration and login tool for web applications using Facebook SSO. The workflow of the prototype enroller is shown in left of Figure 3. (The right side of the figure is the vulnerability evaluator for testing particular vulnerabilities, which we explain in Section 3.1.) Our tool takes a URL as input and determines whether that website uses Facebook SSO. For sites that do, it automatically signs into the site using Facebook test accounts and completes the registration process when necessary. The identity checker attempts to determine if the login was successful. If enrollment succeeded, the checker moves on to the next step of testing for particular vulnerabilities; if the attempt failed, the enroller will continue trying different strategies until either a successful login is detected or a cost threshold is exceeded.

To execute the registration process, the enroller needs to know which DOM element to click and what data to fill in the registration form. Brute-force trials are too expensive to enable large-scale scanning, so our prototype uses heuristics to narrow down the search. Examples of heuristics we have found useful include ignoring invisible or background elements, and ignoring elements that are too small or too large to match sizes commonly used for login buttons. Certain heuristics might work well for most sites, but can be too

[3] It is, of course, possible to automate breaking CAPTCHAs (or hire low-paid labor to manually break many CAPTCHAs [17]). Such techniques, however, raise ethical concerns and questions about the ultimate outcome such an "arms race" that we prefer to avoid.

[4] BugBuster does offer some very limited automatic web application exploration capabilities, but not enough to perform any non-trivial actions such as those involving authentication and business logic without customized application scripts.

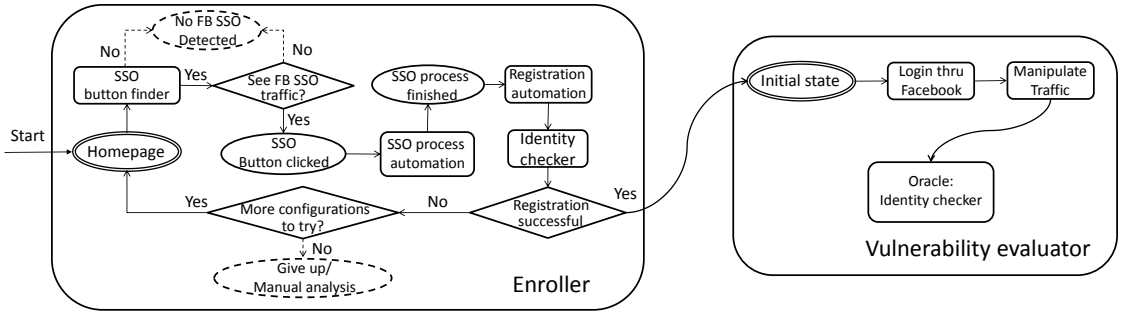

Figure 3: SSOScan overview

restrictive for others. For a particular enrollment attempt, the enroller enables a subset of all available heuristics (which we call a *strategy*, for example, searching for visible buttons only but ignore the element size and layer restrictions) and executes the search.

The purpose of the identity checker is to determine if the session is authenticated and the identity of the user who is logged in. Our Facebook test accounts were generated using personal information that is very unlikely to appear normally. Since we assume most websites will display some user information in authenticated sessions, the identity checker can determine if a user is logged in by examining the pages returned by the application for any strings that match personal information of the test account. Our preliminary experiments support this assumption — the identity checker correctly identifies logged-in users on all but 4% of the tested websites.

Results. To measure the automation success rate, we ran the prototype enroller on the top-ranked 10,000 websites by QuantCast. Of these, the enroller detects 981 sites using Facebook SSO. The enrollment takes an average of around 2 minutes per site to complete. It successfully registers accounts at 785 (80%) of the websites using Facebook SSO. For the 196 sites (20%) where the enroller reported a failure to enroll, we manually inspected each site and categorized the reasons for failure. These provide some of the motivation for our proposed work, as described next.

2.3 Proposed Work: Improving Enrollment

We propose to improve the enroller to enable automated enrollment on more web and mobile applications by exploring several directions for expanding the search space, handling more complex enrollment steps, and improving the efficiency of the search process.

Expanding Search Space. The enroller found 981 websites out of the top 10,000 support Facebook SSO. While we did not manually examine all of the other sites to confirm they had no way to enroll using Facebook SSO, we did look at a sample to confirm that most of the other sites did not appear to support Facebook SSO. We have, however, found some sites where the enroller fails to detect Facebook SSO because the SSO login button is done in a way the does not match any of our string matching heuristics (e.g., an image with no attributes) or is hidden too deeply in the site (e.g., it requires following more than 3 links before reaching it). Of the 196 cases where the enroller does find Facebook SSO, it cannot find the necessary elements to complete the SSO registration process in 25% of the failure cases.

The main reasons these failures are: 1) SSO login button cannot be found because it doesn't match our heuristic string matching (e.g. the login button is an image with no attributes) or the target element is of an atypical element type which the enroller ignores (e.g. a *p* or *li* element); 2) the login button search reaches a maximum depth without finding the SSO button which is hidden under a specific subdomain or resource;

or 3) the enroller is confused and the search is trapped in a local irrelevant area or page for a page that contains multiple login sections.

To address these issues and automate enrollment at more sites, we need to expand the search space. We propose to add more matching string patterns (e.g., non-English labels) and image patters, consider more element types, increase the number of clicks allowed each enrollment attempt before giving up, and expand the maximum size of candidate element list in each search. These improvements to expand search space mostly involve engineering efforts only, however, adding them will increase the testing time per site significantly. This may be tolerable for some purposes such as the developer tool for testing one integration, but will be devastating for conducting large-scale studies (Section 3) without also improving our search strategy, which we discuss next.

Dynamic Strategy Adjustment. Our prototype enroller uses 5 heuristics (examples shown in Section 2.2), which combine to produce 48 possible strategies (one heuristic contains three options). The prototype enroller always starts with the strategy which we think is most likely to succeed based on our previous experiments, and continues through a predefined sequence of strategies. This means there is currently no feedback used to direct the search more efficiently.

As we expand the search space, it is necessary to make this search more efficient by dynamically adjusting the search strategy. For a simple example, currently the enroller attempts to click the top 3 candidate elements on the homepage, and then 3 on each landing page after the first click attempt. This strategy can be dynamically adjusted, based on a per-page score that measures its relevance to Facebook login. The score can be viewed as a progress indicator for the enroller, telling if it has made a move in the correct direction so that it can adjust the upcoming strategy pool size. If the enroller navigates to a high-scoring page, it may expand its strategy pool to attempt to click on more candidate elements on that page. We will investigate how to achieve an effective utility function to compute the current progress.

The enroller can also learn from past failures. We propose to record the attributes of the candidate element and the response or UI changes after the enroller clicks on it. If that enrollment attempt fails eventually, the strategy pool size can be reduced when a future attempt involves clicking on another candidate element of similar attributes, or one that results in a similar response or UI changes. The enroller may also learn from past enrollment successes for other applications. Because of the popularity of third-party helpers (e.g. Gigya, Janrain) and application development frameworks (e.g. Rails and Django), many applications end up with similar code for SSO gadgets and registration forms. By maintaining a database that records information regarding the past successful attempts, the enroller adapt its strategy when familiar elements are found.

Parallelizing and pipelining. Currently the enroller restarts from the initial state when a strategy fails, which is very inefficient. When the enroller perceives a positive progress and is about to expand its strategy pool, it should save the current state as a checkpoint and fork it to multiple browsers to attempt different strategies in parallel. If we observe inconsistencies, an alternative approach would be to save the sequence of actions to achieve the checkpoint state and launch a new browser to follow the same sequence. Once the frontrunner has failed, the backup session can immediately pick up the baton at the checkpoint and attempt a different strategy, which hides the time needed for a complete restart. This can be especially useful when the enroller has taken a long series of actions to finish the SSO login process but fails when exploring to complete the ensuing registration form.

Client-side analysis. Of the top 20,000 websites, 48% implement SSO functionality using JavaScript SDKs or a widget provided by the identity provider. Since the client JavaScript is available to the enroller, we propose to analyze it to identify DOM elements that have event handlers that may trigger the SSO entry function in the SDK, or simply make that JavaScript call directly.

Email verification module. The cause for failure in 5.1% of cases is the need to complete an email verification

step by clicking a link in the verification email (for some sign-on mechanisms like Persona, email verification is an essential step, so this is even more important for other platforms). We propose to address this by adding an additional module to use generated email addresses and to follow links on emails received at those addresses.[5]

2.3.1 Supporting Mobile Platforms

Our preliminary work only supports Facebook SSO on web applications. To expand our coverage, we propose to build support for other SSO services such as Twitter sign-in, Mozilla Persona, and account enrollment on mobile platforms such as Android. Extending the enroller to work with other SSO services mainly requires engineering effort to handle different SSO protocols such as OAuth 1.0a used by Twitter, OpenID used by Google, or browserID used by Persona.

Supporting mobile platforms brings additional challenges as well as opportunities. The major difference between web and mobile applications is that an outside tester will have access to the full mobile application content at client side (e.g. all activities for an Android applications), while the web applications only expose the homepage initially and further contents are delivered upon network requests. This may allow us to quickly locate the GUI elements that trigger the SSO entry point by scanning the client application components for event handlers, especially when mobile OS nowadays has built-in support and standard interfaces to call Single Sign-On services.

We may also need to make changes to the identity checker. We suspect that with a limited display area, mobile applications are less likely to show user information on the screen. However, because the enroller may have access to its memory, internal state or flash storage, it may still be able to pick up such identifying information. We will test this assumption by attempting to enroll a large number of popular mobile applications automatically.

3 Scanning Integrated Applications

This section first describes our preliminary work to build a scanner for detecting vulnerabilities in Facebook SSO implementations (Section 3.1), which builds on the enroller described in Section 2. Then, we discuss our plans to extend scanning to other services and mobile platforms, as well as an alternative human-assisted testing framework (Section 3.2).

3.1 Preliminary Work: Checking SSO Integration

As preliminary work, we developed an automated vulnerability checker for applications integrating Facebook's SSO service. It uses the enroller to obtain an authenticated session and then performs a vulnerability scan that monitors the behavior of the web application under simulated attacks. The prototype implementation checks five vulnerabilities we identified as likely pitfalls in integrating Facebook's SSO service. We provide a brief summary of our preliminary work here; for more details see [54].

Related Work. Integuard [46] and AuthScan [2] are two recent works with similar goals. Integuard infers invariants across requests and responses, and uses this information to perform intrusion detection on future activities. AuthScan is an automated tool to extract specifications from SSO implementations by using both static program analysis and dynamic behavior probing. Unlike these tools, our work focuses on detecting specific vulnerabilities rather than unknown or generic ones. This allows us to establish clear automation goals and build well-defined state machines, and eliminates the uncertainties and inaccuracies that the previous works incur when inferring invariants or modeling unknown functions. We do require

[5] As a by-product of this, we'll be able to identify any sites that are providing these email addresses to spammers if any other emails are received.

knowledge of specific vulnerabilities to check, which may be obtained through the explication process as described in Section 1.

Vulnerability Evaluator. The goal of the Vulnerability Evaluator is to simulate attacks and determine if the application enters a compromised state. The right side of Figure 3 illustrates the key steps. After the enroller succeeds in creating two test accounts at target website, it passes control to the Vulnerability Evaluator, which resets the applications state by deleting all the cookies. Then, it follows the same action recorded by the enroller to initiate the SSO process again, but meanwhile manipulates the traffic to simulate several attacks on the tested site and records the responses. Finally, it consults an oracle to automatically determine if the recorded interactions reveal any of the tested vulnerabilities.

The goal of the oracle in SSOScan is to obtain the identity of the current authenticated session. It is implemented using the enroller's identity checker. The oracle checks the homepage response for identifying information from either test accounts. When user A's login credentials were provided, but after traffic manipulation the resulting response contains information from user B or both users (application in a confused state, which we have observed for several tested application), the oracle reports that the application is vulnerable.

Evaluation. Figure 4 summarizes our results. We tested the scanner on the top-ranked 20,000 sites according to QuantCast [15]. Of these, it found Facebook SSO on 1700 could be exploited via at least one of the five tested vulnerabilities. Our prototype implementation can detect whether a target application contains any of the vulnerabilities with an average testing time of 3.5 minutes.

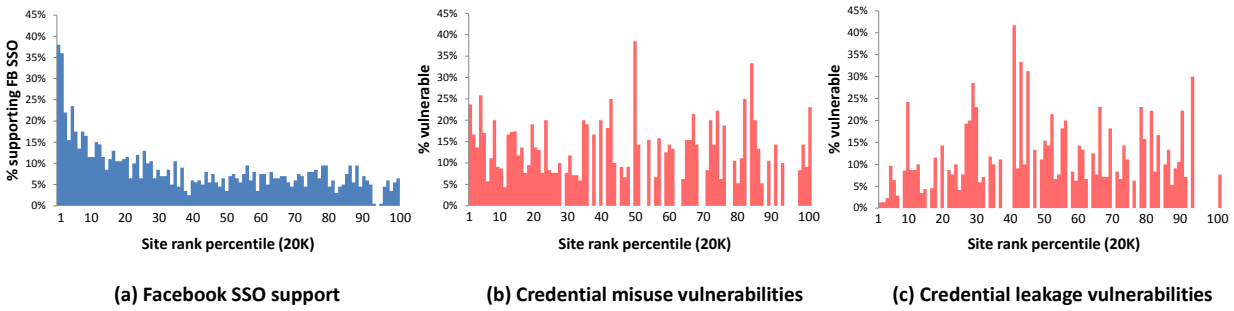

Figure 4: Facebook integration results by site ranking

Each bin in figure (a) contains exactly 179 sites, 1% of the total tested. Each bin in figures (b) and (c) reflects the percentage of vulnerable sites over all site that implements Facebook SSO in that ranking range (which varies across bins).

3.2 Proposed Work

Our results with the prototype scanner provide support for the claim that it is possible to automate deep web security scanning for many typical websites, and that doing such testing frequently reveals important security vulnerabilities. We propose to extend the vulnerability scanner to check other services, as well as to support checking mobile application (Section 3.2.1). The other main direction is to enable human-assisted scanning for vulnerabilities that involve interactions too complex to automate completely (Section 3.2.2). This will not be suitable for fully automated large-scale scanning experiments, but expecting some assistance from developers is reasonable when the scanner is deployed as a stand-alone service.

3.2.1 Checking Mobile Applications

Our preliminary work focused on web applications, but mobile applications are increasingly being used for security-sensitive transactions. Extending our approach to also support mobile applications involves

both engineering work and some interesting new challenges. In addition, the app market model provides a good deployment opportunity since the vulnerability scanner could be used to test all submitted apps. Although our approach applies to any mobile platform, we focus on Android because of its widespread use, open source basis, and more open app market model.

One particular target of our mobile app scanning will be in-app purchase services offered by leading marketplaces such as Google Play and Amazon Appstore that are used by many top-ranked applications. Typical in-app purchases are initiated when a user elects to purchase additional downloadable content in the application and is directed to an app store interface where she logs in and confirms the purchase. After receiving the payment, the app store issues a *purchase token* as the receipt to the user's device. The application is responsible for forwarding the token to its back end server which should verify the receipt and deliver the content to the user's client. Application developers often make mistakes in implementing this (especially the last step), as demonstrated by iAP Cracker [19], which exploits this to allow device owners to obtain in-app content for free.

We propose to develop a tool that automates logging into the application using technique similar to the web enroller, and then to simulate a user exploring the app to find opportunities for in-app purchases. This can be directed by code analysis given that the entire client application is accessible to the scanner, similar to what we have proposed in Section 2.3. To test for vulnerabilities, we will use an OS-level proxy to intercept the purchase request and response messages and replace them with simulated attacks, and an oracle that determines if the purchase successfully completed.

The mobile enroller can also be combined with previous developer frameworks for application testing and information flow tracking. For example, it could be added to existing GUI testing tool [52, 34] to improve its execution space exploration or code coverage. To measure sensitive information leakage, we could integrate the enroller with systems that support taint tracking such as TaintDroid [20] or OS level instrumentation.

3.2.2 Human-Guided Testing

Automatic registration will be almost impossible to achieve in certain scenarios, such as when the service implements a two factor authentication or requires application-specific information such as an invitation code. The vulnerability evaluator will also be difficult to automate for testing complex interactions such as those required for payment APIs. This may require automatically adding items to cart and checking out with a credit card, which varies drastically from application to application (and could be dangerous to attempt to automate).

We propose to address these by using assistance from humans to provide guidance to the scanner. We will build a tool for developers, implemented as a browser extension, that developers can use to guide the scanner through interaction sequences. Developers would use this tool to follow the normal login or registration process in their web browser, perform necessary user interactions to invoke the third-party service based on specific vulnerability. The tool would be integrated with the testing oracle to automatically detect vulnerabilities by simulating attacks, and it will also record the necessary user interaction sequence to be able to replay them for future testing.

To minimize developer effort and maximize coverage, we will generalize the recorded action sequence into one that can replayed and varied in scanning. When recording the trace initially, the developer-assisted framework will also instrument the application and record the actual execution path of the test. For web applications, the trace could be a sequence of pages visited and functions called to generate the dynamic page contents; for Android applications, it can be a sequence of activities and methods invoked. Afterwards, the framework can attempt to vary the initial user interaction sequence into other sequences that follow a similar execution path by shuffling and randomizing different user interactions, and by using program analysis techniques such as event-space constraint guided symbolic execution [51]. The generated execution traces may be especially helpful when developers release a new version of the application with significant changes

in either its user interface or core logic code, but not both. The vulnerability scanner can quickly update either the user interaction sequences or execution paths using the one which remains largely unchanged, and keep them both up-to-date with the new version of the application.

4 Education, Outreach, and Transfer Plans

The proposer has a strong commitment to education, outreach, and to making useful tools available to the research community, government, and industry.

Graduate and undergraduate student research. The proposer views mentoring research students as among his most important responsibilities and has a strong record and serious commitment to producing PhD students who go on to successful careers in academia, government labs, and industry, as well as a successful record for developing undergraduate researchers who go on to earn PhDs at top institutions. Seven undergraduates supervised by the proposer have been recognized by the national *CRA Outstanding Undergraduate Research Award* (including one runner-up and two finalists). Two undergraduate students are already contributing to the preliminary work which forms the basis of this proposal, and the proposed work is well-suited to introducing students to research at a variety of levels from straightforward systems-building experience extending our approach to new platforms to more challenging research developing new formal techniques and search strategies.

Student and professional development. This work is largely motivated by the inability of developers to build secure systems — our proposed work on explicating SDKs aims to make explicit the assumptions needed for security that developers without security expertise may not understand, and our automatic scanning work is designed to automated tests for certain security vulnerabilities. Nevertheless, building secure systems essentially depends on developers appreciating the challenges and understanding security fundamentals. The PI incorporates security into all his courses, and is developing a new operating systems course that emphasizes building robust and secure systems (`rust-class.org`). In addition to his university courses, the PI has taught an open on-line applied cryptography course that was ranked as the #1 online class for building IT careers [12] and regularly teaches security and applied cryptography courses to working engineers (one recent series of courses generated sufficient interest to be moved to a local movie theater [24]). Such education efforts are important for reaching working programmers who build the systems we hope to make more secure.

Tool Distribution. Results from this work will be released as open source tools, including the automated scanning tools and the models resulting from our explication experiments. In addition to releasing the source code, we will also make our checking tools available to developers as web services to make them as easy to get started with as possible (our preliminary prototype, `ssoscan.org`, gives a glimpse into what we have in mind for these). The PI has a long history of releasing and supporting open source tools including the Splint lightweight static analysis tool (incorporated in most Linux distributions) and the FastGC secure computation framework and MCL simulator, both of which are widely used in academia and industry.

5 Results from Prior NSF Support

The proposer has been the PI for 6 completed, and one ongoing, NSF research grants. Research funded by these grants has resulted in publications that have been cited over 4000 times, produced software tools that are incorporated into several commercial products and most major Linux distributions and funded undergraduate research students who have gone on to PhD programs at CMU (4), UC Berkeley, Stanford, U. Washington, UCLA, and others. The most relevant previous grant is *CPA: Automatic Inference and Effective Use of Temporal Properties* ($330K, 7/15/06-6/30/09, 0541123).

Intellectual Merit. This project developed dynamic inference techniques for mining properties from im-

perfect traces. New ideas developed by the work include a hierarchy of property templates used to infer dynamic properties, new statistical techniques for enabling property inference on the kinds of imperfect traces found in practice, and a comprehensive study the demonstrated the effectiveness of both the inference and analysis techniques on several industrial programs including OpenSSL and the Windows kernel.

Broader Impacts. The research led to publications [49, 48, 50, 47] with over 300 citations, and the Perracotta tool (`http://www.cs.virginia.edu/perracotta/`), which has been used for industrial kernel development at Microsoft as well as in several academic projects. The project produced one PhD graduate, and two undergraduate researchers involved in the project went on to top PhD programs.

6 Impact Summary

Our work is driven by the observation that security exploits succeed because an adversary finds a way to break some assumption upon which security depends. We seek to develop a way to make those assumptions explicit by using a systematic and principled method for uncovering implicit assumptions in third-party service SDKs (Section 1), and to provide tools that can automatically test applications for common pitfalls that allow those assumptions to be violated (Sections 2 and 3). We will evaluate our explicating process on a variety of SDKs, conduct large-scale studies using the vulnerability scanners, and provide open and ready-to-use tools for developers.

The major milestones for the project are summarized below.

Year 1

- Develop techniques to automatically probe remote services to build semantic models.
- Investigate program slicing and taint tracking techniques to automate model extraction.
- Invent flexible heuristics to improve the enroller's effectiveness.
- Adapt the enroller to support Android applications.
- Evaluate approaches to dynamically adjust the search to reduce per-site testing time.

Year 2

- Evaluate the improved explication process by applying it to several third-party web and mobile services.
- Incorporate analysis of available source code to complement heuristics search.
- Develop a checkpoint-enabled parallel testing framework to enable larger-scale tests.
- Conduct a large-scale vulnerability scanning study on Android applications.
- Develop the browser extension for recording user interactions.

Year 3

- Develop the payment SDKs (including in-app purchases) vulnerability scanner, address the challenges in generating the necessary user interactions and building oracles for both web and mobile applications.
- Develop the human-assisted vulnerability scanner and investigate methods for effectively varying the recorded action sequences.
- Evaluate the vulnerability scanners by scanning a large number of popular Android applications.
- Evaluate the human-assisted vulnerability scanner by manually performing the human guidance for a set of test applications and measuring how effective the techniques for varying interactions are at uncovering vulnerabilities.