SVAuth – A Single-Sign-On Integration Solution with Runtime Verification

Shuo Chen[1], Matt McCutchen[2], Phuong Cao[3], Shaz Qadeer[1], Ravishankar K. Iyer[3]

[1] Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

[2] Massachusetts Institute Of Technology
Cambridge, MA 02139, USA

[3] University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Abstract. SSO (single-sign-on) services, such as those provided by Facebook, Google and Microsoft Azure, are integrated into tens of millions of websites and cloud services, just like lock manufacturers offering locks for every home. Imagine you are a website developer, typically unfamiliar with SSO protocols. Your manager wants you to integrate a particular SSO service into a website written in a particular language (e.g., PHP, ASP.NET or Python). You are likely overwhelmed by the amount of work for finding a suitable SSO library, understanding its programming guide, and writing your code. Moreover, studies have shown that many SSO integrations on real-world websites are incorrect, and thus vulnerable to security attacks! SVAuth is an open-source project that tries to provide integration solutions for all major SSO services in all major web languages. Its correctness is ensured by a technology called self-verifying execution, which performs program verification at runtime. SVAuth is so easy to adopt that a website developer does not need any knowledge about SSO protocols or implementations. This paper describes the architecture of SVAuth and how to use it on real-world websites.

1. Introduction

SSO (single-sign-on) services, such as those provided by Facebook, Google and Microsoft Azure, are integrated into tens of millions of apps, websites and cloud services, just like lock manufacturers offering locks for every home. However, the integration practice is very ad-hoc: on one hand, protocol documentation and usage guides of SSO libraries are written by experts, who are like experienced "locksmiths"; one the other hand, most website programmers are not "locksmiths", and inevitably fall into many pitfalls due to misunderstandings of such informal documentation. Security bugs in SSO integrations are continuously discovered in the field, which leave the front door of the cloud wide-open for attackers. SSO bugs are the primary examples when the Cloud Security Alliance ranked API integration bugs as the No. 4 top security threat [10]. These bugs have become a familiar theme in academic conferences [1][4][7][15][17][18][20] and Black Hat conferences [7][19].

We are working on an open-source project, called SVAuth, to provide every website with a SSO integration fundamentally immune from the aforementioned

bugs. Moreover, we try to make SVAuth a very easy and widely applicable solution for real-world adoption: (1) it is language independent, so it works with web apps in any language, such as PHP, ASP.NET, Python; (2) the default solution only requires installing an executable (or simply copying a folder), without any library integration effort; (3) a programmer can customize the default solutions for his/her special requirement. The customized solutions enjoy the same correctness assurance as the default ones; (4) the SVAuth framework can accommodate all SSO services.

The correctness of every SSO solution in SVAuth is assured using our published technology called self-verifying execution (SVX) [14]. The basic idea of SVX is to perform program verification on a per-execution basis. In this sense, SVX is a type of *runtime verification* approach. We will briefly describe how the SVAuth's object-oriented framework incorporates the SVX verification technology. This ensures that a correctness property defined at the most abstract level is satisfied in every execution of every concrete website implementation.

Besides explaining the basic idea, the paper focuses on the source code architecture of SVAuth, and provides a walk-through on how to use SVAuth in various scenarios. We will also show the integration of SVAuth with MediaWiki and HotCRP as examples.

Secure programming of web-based protocols is a nice problem space to apply runtime verification technologies like SVX. We hope programmers and researchers will join the SVAuth effort by contributing code to the project or by adopting the SSO solutions for their websites.

2. The SVAuth framework

This section explains the design and underlying technology of SVAuth. Readers who only want to use SVAuth can skip this section and jump to Section 3.

SSO is a user authentication mechanism commonly seen on websites. For example, NYTimes.com's login page has a button "Log in with Facebook". When a user clicks the button, through an SSO protocol, Facebook will convince NYTimes.com that the visiting user is a particular user recognized by Facebook. In SSO's terminology, Facebook is called the identity provider; NYTimes.com is called the relying party.

SVAuth provides a solution for relying parties to easily and securely integrate SSO services provided by identity providers. The code of SVAuth is publicly available at https://github.com/cs0317/svAuth. Figure 1 illustrates the class hierarchy of the code, written in C# targeting the .NET Core runtime [1]. It has four levels. The first level defines the most generic concepts, like identity provider (IdP), relying party (RP) and authentication conclusion. More importantly, it defines a correctness property φ to capture the intrinsic meaning of "an RP's conclusion about the client's identity is correct". Defining φ is a non-trivial job, involving fairly deep understandings about the notion of "authentication". We skip the details of φ in this paper. Readers can consider φ as a property that all SSO systems should maintain.

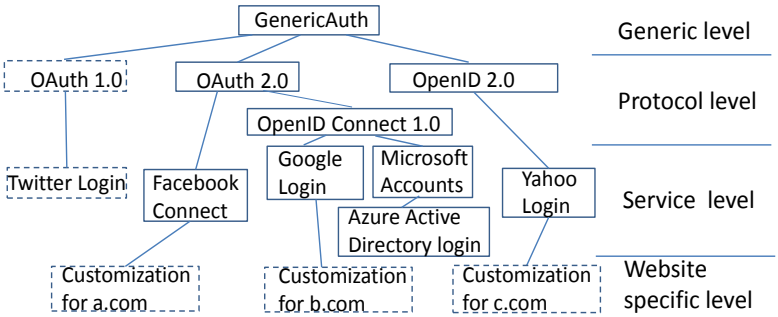

Figure 1: The class hierarchy of SVAuth

The protocol level consists of classes derived from the generic level. These classes match the protocol specifications, such as those for OAuth 2.0 and OpenID Connect 1.0. Note that the boxes with dashed lines represent namespaces that we have not yet implemented. At the service level, more concrete classes are defined to implement SSO functionalities provided by various companies. There are over 30 major companies in the world providing SSO services. SVAuth has implemented 7 namespaces, each covering one service. We are expanding the coverage, and hope to support most SSO services. At the website specific level, a developer can choose to customize the SSO system by deriving classes from the upper levels in order to fit the specific need of a website.

The verification goal. Having this class hierarchy, the verification goal of the SVAuth framework is: *every concrete implementation in SVAuth should satisfy property φ defined at the generic level*. Next, we briefly explain the self-verifying execution approach to achieve this goal.

2.1. Self-verifying execution (SVX)

The basic idea of self-verifying execution (SVX) is to perform code verification at runtime on a per-execution basis. The details are described in our earlier paper [14], which explains the advantages of SVX and how it works. Below, we use a simple example to explain the technology.

Imagine there are three *collaborative* websites, Alice.com, Bob.com and Charlie.com. Each website has an integer constant. They want to run a web-based protocol (i.e., a protocol driven by a browser) to determine which website holds the largest integer. Alice.com has a public method grab() to return Alice's integer value; Bob.com has a method compare(m) to compare its own value against the value in the input message m, and return the larger one; Charlies.com has a method finish(m) to compare its own value against the value in the input message m, and calls a local method conclude(conc). The correctness property φ is that: whenever conclude(conc) is reached, conc.value should be the largest value and conc.who should indicate the website holding the value.

Figure 2 shows two executions when the three websites hold values 10, 40 and 5 respectively. The left execution is expected by the programmer, and it results in

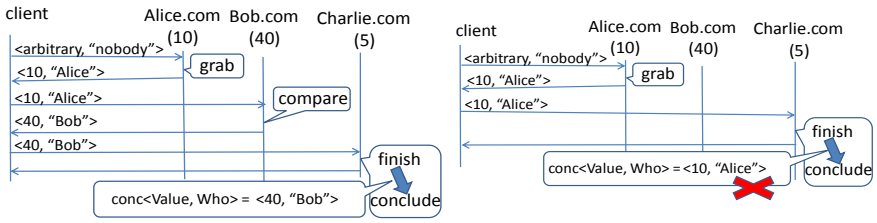

Figure 2: An expected execution (left) and a successful attack (right).

the correct conclusion <40, "Bob">. However, the system is actually vulnerable because a malicious client can trigger an execution on the right, which results in <10, "Alice">. The traditional goal of program verification is to find such attacks and reject the implementation. However, as we explained in the earlier paper [14], this kind of verification requires the programmer to build a precise model for the client, and the theorem to prove has to be inductive, because the client is basically an infinite loop to interact with the websites.

SVX is much simpler. It does not require the modeling of the client. Instead, it lets a real client trigger an actual execution. During the execution, SVX records the method sequence being executed. In the end of each execution, it only proves that *this method sequence* satisfies property φ, rather than that *every possible execution* will satisfy property φ. Therefore, the theorem to prove is much simpler; it usually requires no induction and can be proven fully automatically.

Figure 3 shows the SVX-enhanced execution. The only addition of SVX is the third message field, which is called SymT (symbolic transaction). In SymT, #grab, #compare and #finish are hash values automatically computed over the code of the corresponding methods using the reflection capability of the language. The hash values represent the unique semantics of these methods. To determine whether a conclusion is correct, we only need to prove that the final SymT logically implies φ. For example, Charlie.com:#finish(Bob.com::#compare(Alice.com::#grab())) implies φ, so <40, "Bob"> is correct and thus accepted. On the other hand, the previous attack sequence Charlie.com:#finish(Alice.com::#grab()) does not imply φ, so the conclusion <10, "Alice"> is incorrect, so it is rejected at runtime.

The detailed specification of the SymT is given in Section IV.A in reference [6]. A SymT captures the facts whether a message is *signed* or *unsigned*, and whether it

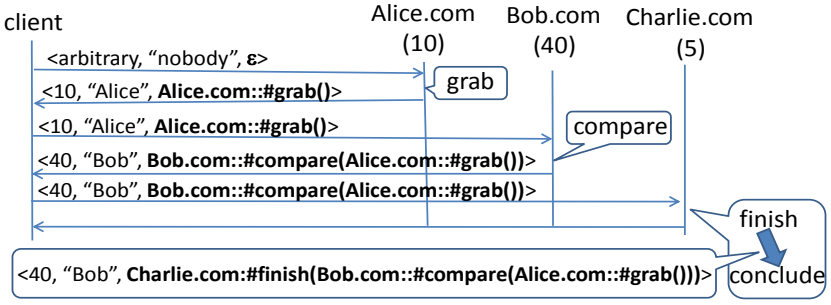

Figure 3: A concrete execution of SVX.

is *a server-to-server call* or a *browser-redirection*. These determine whether the output of each method call was authenticated as coming from a known party or should be treated as untrustworthy and nondeterministic in our verification. Due to space constraints, we skip the details here.

The current SVX library uses an off-the-shelf C# verifier, consisting of Bytecode Translator [5] and Corral [12] (see [14] for details).

Runtime overhead of SVX. For every execution, SVX proves a theorem corresponding to the executed code sequence, so the runtime overhead seems too high to be practical. However, since the SymT only represents the code, not any concrete data value, every theorem can be effectively cached (by Charles.com in our example). Therefore, the runtime overhead of SVX is near zero, unless an attacker triggers a new execution sequence or the source code is revised.

2.2. Incorporating SVX into the class hierarchy

Recall that the verification goal of SVAuth is to ensure all *concrete* implementations to satisfy the property defined at the *generic* level. A significant advantage of SVX is that the self-verifying capability can be inherited automatically, thus scaled up to all concrete systems. In SVAuth, only the upper two levels, i.e., the generic level and the protocol level, are aware of SVX. Programmers at the lower two levels only need to do normal OO inheritance, and every execution on every concrete implementation will be verified.

It is worth pointing out that this advantage does not exist in static verification techniques. There is a well-known dilemma that a property statically proven for a base class may not hold for a derived class, which is usually discussed in the context of the Liskov Substitution Principle (LSP) [13]. The dilemma of LSP in real-world scenarios is often explained using the "Rectangle-and-Square" example [11].

3. Using SVAuth on Websites

Section 2 explains the SVAuth framework and the underlying verification technology. None of the knowledge is needed for a website programmer to use SVAuth. This section explains how to deploy SVAuth. From the deployment standpoint, SVAuth consists of two components, shown in Figure 4:

(1) **Agent**: this is the C# code implementing the class hierarchy in section 2, with the SVX capability built in. The agent runs on the .NET Core runtime as an executable, listening on its own port. It bundles a verification toolchain that currently depends on .NET Framework and thus only runs on Windows, but we see no fundamental obstacle to porting the verification toolchain to .NET Core or adding support for the agent to use a remote verification service running on Windows.

(2) **Adaptor**: a language-specific but protocol-agnostic component that is added to the RP website, communicates with the agent and makes the authenticated user identity available to the RP web application framework. For example, we have a PHP adaptor, an ASP.NET adaptor, etc.

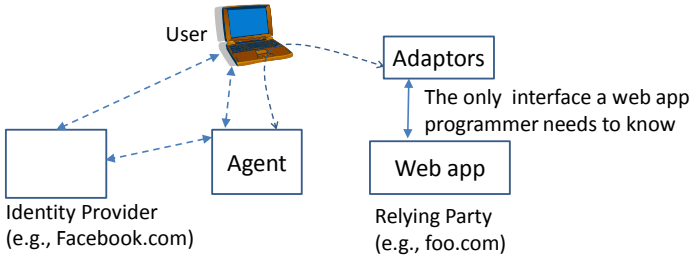

Figure 4: Agent and adaptors.

The agent handles complex interactions represented by the dashed lines: suppose the identity provider is Facebook.com and the web app is written in PHP, then the agent's job is to authenticate the user through Facebook.com, and call the PHP adaptor to set the authentication conclusion into a set of PHP session variables. However, the web app programmer does not see any of these interactions, but only need to remember the following simple interface:

(1) "http://foo.com/SVAuth/adaptors/**php**/start.**php**?provider=**Facebook**" is the URL to start the Facebook sign-on;
(2) The information about the authenticated user will be in the session variables, such as SVAuth_UserID, SVAuth_Email, SVAuth_FullName, etc.

3.1. Three scenarios for deploying the agent

Figure 4 does not explicitly show where the agent runs. There are three possible scenarios for the agent's placement.

Public agent. In the GitHub repository of SVAuth, we have a release of the adaptors. Using this release is the simplest way to incorporate SSO logins, because it by default uses a public agent running on port 3020 of server *https://authjs.westus.cloudapp.azure.com*. The web app programmer's job is extremely simple: just copy the released *adaptors* folder into the website directory *http://foo.com/SVAuth*.

Local agent. If a programmer does not want to use the public agent, perhaps because he/she wants to derive some customized classes in the agent, then setting up a local agent is an option. A local agent runs on the same server as the web app, i.e., foo.com. The programmer's job is also simple: just do "git clone" the SVAuth repository onto the foo.com server. The "git clone" command will pull the whole SVAuth code, including the agent, the adaptors, as well as the dependencies BCT and Corral. To run the agent, install the .NET Core runtime, and use the command "dotnet run" inside the folder */SVAuth/SVAuth* of the cloned repository. By default, the adaptors know that the agent runs on port 3000 of the local server.

Private agent. A private organization can set up a server to run the agent, which serves all websites inside the organization. For example, the company owning the

domain *foo.com* can set up a server *SVAuth.foo.com* to run the agent. The programmer uses "git clone" to get the code onto the *SVAuth.foo.com* server, following the steps described in the "local agent" scenario. Then, the programmer of every website follows the steps described in the "public agent" scenario, except that the configuration file of the adaptors should specify "*SVAuth.foo.com*" as the agent and "*.foo.com*" as the agent's scope.

3.2. Integrating SVAuth with real-world web apps

The interface between the web app and the adaptors is really simple, as described earlier. All the web app programmer needs to do is to read user's identity data from the session variables. This is independent of the SSO service and the language of the web app. To show the ease of the integration, we have integrated SVAuth with MediaWiki and HotCRP. MediaWiki is the software powering Wikipedia.org, and HotCRP is a sophisticated conference management system. The following two URLs are the two systems with SVAuth integrated:

The MediaWiki demo: http://authjs.westus.cloudapp.azure.com
The HotCRP demo: http://authjs.westus.cloudapp.azure.com:8000/

The integrations were very easy. The MediaWiki integration only needs 8 lines of code changes to the MediaWiki user login plugin. The HotCRP integration has only 21 lines of changes. These integrations are documented in the "IntegrationExamples" folder in the code repository.

Inside Microsoft Research, a MediaWiki website has been using SVAuth for nearly a year, which authenticates users through Microsoft Azure Active Directory.

3.3. Currently supported SSO services and web languages

The SSO services currently supported by SVAuth are: Facebook, Microsoft, Microsoft Azure Active Directory, Google, Yahoo, LinkedIn and Weibo (a major identity provider in China). The demo page for these services is at http://authjs.westus.cloudapp.azure.com/SVAuth/adapters/php/AllInOne.php. The currently supported languages are: PHP, ASP.NET and Python. We are expanding the coverage for SSO services and web languages.

4. SVX and SVAuth in the context of runtime verification

We believe that SVX and SVAuth bring some new perspectives to the research community. In this section, we discuss them in the context of runtime verification.

SVX's relation to other runtime verification techniques. Similar to other runtime verification techniques, SVX performs the actual verification at runtime on a per-execution basis. The SymT string is a representation of an execution trace, and SVX is able to verify *safety* properties about traces. Relations can be drawn between SVX and Monitoring-Oriented Programming (MOP) [9]. MOP is a generic framework for programmers to specify relevant events and safety properties. The

MOP framework automatically instruments the code to monitor these events, maintain monitored states and verify the safety properties. It will be valuable to think how SVX can utilize the automatic mechanisms in MOP. Currently, SVX requires the programmer to manually call the SVX library to compute the SymT (analogous to MOP's event monitoring) and to verify a property (analogous to MOP's property checking). In addition to automation, another advantage of MOP is to separate the monitoring logic from the logic of the monitored program.

SSO and other potential domains for runtime verification. SSO is a good domain for runtime verification, because (1) it is a problem faced by many programmers, and its engineering practice is ad-hoc today; (2) it involves multiple organizations: protocol working groups, identity provider companies, SDK solution providers and website programmers, who work collaboratively across different abstraction levels, as shown in Figure 1; and (3) the desired safety assurances can be defined as trace properties. Thus, SVAuth, which builds SVX into the OO class hierarchy, is a suitable solution.

Besides SSO, *user authorization* can be another problem domain. Mobile apps, social sharing web apps and IoT devices all require proper user authorization. It meets the above conditions (1) and (2). A new challenge is that the authorization objectives are often much more diverse than those for SSO. A research question is: how to precisely define these diverse properties (by individual programmers on a case-by-case basis)?

Also, SVAuth is an instance of "protocol spec as code". This concept has been put in practice in other areas. For example, the TPM (Trusted Platform Module) 2.0 library specification [16] is primarily C code with a substantial amount of English comments (unlike most of today's protocol specs, written in English with pseudo code samples as "comments"). It is valuable for the runtime verification community to identify other areas where industrial specifications can be written as code (e.g., like the abstract classes in SVAuth). The specifications can be for distributed systems, device drivers, online payments, and IoT (e.g., the AllJoyn framework [2]).

5. Summary

Integrating SSO services is often seen as a non-trivial programming job, which demands expertise and time. Moreover, the current practice is too ad-hoc, and SSO vulnerabilities are so pervasive that they have become a trendy topic in security conferences. We want to promote the SVAuth solution – it provides a higher correctness assurance, and is an easy solution for websites to integrate SSO.

The underlying technology is self-verifying execution (SVX), which is a runtime verification mechanism. It is combined with the OO class hierarchy to form the framework for all concrete implementations to be verified with respect to a generic property defined over the base classes. SVAuth is extensible, so we hope researchers working on verification technologies contribute to the core components of the project, or build more SSO solutions within the SVAuth framework.

References:

[1] .NET Core. https://www.microsoft.com/net/core

[2] AllJoyn® Framework. https://allseenalliance.org/framework

[3] Chetan Bansal, Karthikeyan Bhargavan, Sergio Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. 25th IEEE Computer Security Foundations Symposium, CSF 2012

[4] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong, Authscan: Automatic extraction of web authentication protocols from implementations. ISOC Network and Distributed System Security Symposium, 2013

[5] Michael Barnett and Shaz Qadeer. "BCT: A translator from MSIL to Boogie." Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2012.

[6] Eric Chen, Shuo Chen, Shaz Qadeer, and Rui Wang. Securing Multiparty Online Services via Certification of Symbolic Transactions, *IEEE Symposium on Security and Privacy*, 2015.

[7] Eric Chen, Yutong Pei, Yuan Tian, Shuo Chen, Robert Kotcher, Patrick Tague. 1000 Ways to Die in Mobile OAuth. Blackhat USA 2016.

[8] Feng Chen and Marcelo d'Amorim and Grigore Rosu. Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP. The Runtime Verification Workshop 2005

[9] Feng Chen, Grigore Rosu. MOP: An Efficient and Generic Runtime Verification Framework. The ACM SIGPLAN conference on Systems, Programming, Languages and Applications (OOPSLA) 2007.

[10] Cloud Security Alliance. "The Notorious Nine – Cloud Computing Top Threats in 2013". https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf

[11] DZone.com. "The Liskov Substitution Principle (With Examples)". https://dzone.com/articles/the-liskov-substitution-principle-with-examples

[12] Akash Lal, Shaz Qadeer and Shuvendu Lahiri. "A Solver for Reachability Modulo Theories". Computer Aided Verification, 2012

[13] Barbara H. Liskov, Jeannette M. Wing. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16 (6): 1811–1841. November 1994.

[14] Matt McCutchen, Daniel Song, Shuo Chen, Shaz Qadeer. Self-Verifying Execution (Position Paper). Proceedings of the IEEE Cybersecurity Development Conference (SecDev), 2016

[15] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. ACM conference on Computer and Communications Security 2012.

[16] Trusted Computing Group. Trusted Platform Module Library Specification 2.0. http://www.trustedcomputinggroup.org/resources/tpm_library_specification

[17] Rui Wang, Shuo Chen, XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. *IEEE Symposium on Security and Privacy*, 2012.

[18] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. *USENIX Security*, 2013.

[19] Ronghai Yang, Wing Cheong Lau, Tianyu Liu. Signing into One Billion Mobile App Accounts Effortlessly with OAuth 2.0. Blackhat Europe 2016. .

[20] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. *USENIX Security*, 2014