Self-Verifying Authentication – A Framework for Safer Integrations of Single-Sign-On Services

Shuo Chen, Shaz Qadeer, Matt McCutchen, Phuong Cao, Ravishankar Iyer
Microsoft Research, Massachusetts Institute of Technology, University of Illinois
Motivation

• SSO – the “front door” lock for tens of millions of websites
  • E.g., Airbnb.com allows Facebook sign in.

• Many companies provide identity services
  • Provide SDKs (i.e., lock products) for different web languages
  • Step-by-step instructions to teach programmers
    • E.g., OpenID Connect 1.0 spec, Azure AD dev guide

• But most website programmers are not experienced “locksmiths”
  • Imagine that you need to read an installation sheet, drill holes, and
    install a lock cylinder, knobs and steal plates on your front door
  • Can every average homeowner do it securely?
Security-Critical Logic Bugs are Pervasive

• Numerous studies have shown serious bugs
  • Papers in leading academic security conferences
  • Findings from the Black Hat community
    • E.g., in Black Hat USA 2016 and Black Hat Europe 2016

• Consequences:

<table>
<thead>
<tr>
<th>Login safety</th>
<th>An attacker can sign into a victim’s account</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login intent</td>
<td>A victim can be tricked to sign into an attacker’s account (login forgery - CSRF)</td>
</tr>
</tbody>
</table>

• Cloud-API integration bugs are the No.4 cloud security top threat
  • SSO logic flaws are the primary example of this bug category
• Demo 1:
  • Microsoft Azure AD library for Node.JS
  • **Login safety violation**: attacker logs into any victim’s account
  • Video

• Demo 2:
  • [https://web.skype.com](https://web.skype.com)
  • **Login intent violation** via request forgery: victim unknowingly login into the attacker’s account
  • Video1  Video2

• We have reported many SSO issues to various identity providers and websites.
  • Companies, big or small, make these mistakes.
Example: an SSO bug due to insufficient logic checks using Google ID

A simplified illustration of the Google ID protocol

- In 2012, it was based on Open ID 2.0

```
redirection1:
  realm= the RP’s identity
  required=(email,firstname,lastname)

redirection2:
  signed=(email,firstname,lastname)
  email=“alice@a.com”
  firstname=“Alice”
  lastname=“Smith”
  signature=“HRU436ETQ95TR939”
```
Vulnerability and attack

redirection2:
signed=(firstname,lastname)
signature="HRU436ETQ95TR939"
firstname=“Bob”
lastname=“Johnson”
email=“alice@a.com”

redirection2:
signed=(firstname,lastname)
signature="HRU436ETQ95TR939"
firstname=“Bob”
lastname=“Johnson”
email=“alice@a.com”

redirection1:
realm= the RP’s domain
required=(email,firstname,lastname)

Google’s signature verified.
Welcome, user “alice@a.com”!

redirection1:
realm= the RP’s domain
required=(email,firstname,lastname)

Google’s ID service

Bob’s browser

Relying party website
Example: unintended usage of OAuth 2.0 access token

LiveID OAuth Identity Service

(1) WL.login (“wl.basic”)

(2) token

(3) me(token)

(4) Alice’s basic info

FooApp on Alice’s device

Foo.com service

Welcome, Alice

Welcome, Alice
Confusion about authentication and authorization

ID Office

The President authorizes everybody to view his public photo

Hi, I want to see the photo

Here is the token you need

OMG!
He (-President) is the President!

Check out my photo with this token.
Program verification to prevent logic bugs in SSO

Our verification technology: self-verifying execution (SVX)
Hurdles of traditional verification approaches

• Why can’t I feed my source code $P$ and a property $\varphi$ into a program verifier, and expect bugs to be found automatically?

• Because program verification is a very challenging task
  • Need to model the runtime system $R$ – hard to be precise
  • Need to model the unknown attacker $A$ – hard to be exhaustive
  • Theorem to prove: if attacker $A$ calls $P$ for infinitely many times, and each time has multiple public APIs, can $\varphi$ ever be violated?
    • Need to prove by induction (because of the infinite possibilities of executions) – hard to automate.

Hypothetical Attacker $A$

Program $P$ of interest

underlying runtime system $R$

Safety property $\varphi$ violated?
Basic idea of SVX

• Every actual execution is responsible for collecting its own executed code, and proving that it satisfies $\varphi$.

• No need to model the attacker
  • Because every execution is driven by a real user.

• No need to model the runtime platform
  • Because execution happens on the actual platform

• No need for inductive proof
  • Because it only proves “this execution satisfies $\varphi$”, not “all possible executions satisfies $\varphi$”.
Distributed consensus: comparing integer constants among three websites

Alice.com

const int Value = 10;
Message grab (Message m1) {
    m2 = <Value, “Alice”>;
    m2.SignBy(“Alice.com”);
    return m2;
}

Untrusted client

Safety property $\varphi$:
Whenever $\text{conclude}(m2)$ is reached, $m2$ must represent the website holding the biggest int.

Bob.com

const int Value = 40;
Message compare (Message m1) {
    ValidateSignature(m1);
    Message m2;
    m2 = <Value, “Bob”>;
    m2 = max(m1,m2);
    m2.SignBy(“Bob.com”);
    return m2;
}

Charlie.com

const int Value = 5;
Message finish (Message m1) {
    ValidateSignature(m1);
    Message m2;
    m2 = <Value, “Charlie”>;
    m2 = max(m1,m2);
    conclude(m2);
    return m2;
}
The expected protocol flow

- Alice.com
  - (10) Alice
  - (10) Alice
  - (40) Bob
  - (5) Charlie

- Bob.com
  - (40) Bob

- Charlie.com
  - (5) Charlie

- Client
  - <arbitrary, “nobody”>
  - <10, “Alice”>
  - <10, “Alice”>
  - <40, “Bob”>
  - <40, “Bob”>

Actions:
- grab
- compare
- finish
- conclude
The system is vulnerable!

client

<arbitrary, “nobody”>

<10, “Alice”>

Alice.com (10)

grab

Bob.com (40)

Charlie.com (5)

<10, “Alice”>

finish

conclude

<10, “Alice”>
How SVX works

• Attach a field, namely SymT (Symbolic Transaction) onto every message.
• #grab, #compare and #finish are a compact representation of the executed code of these methods.
Verifying an execution

• Method conclude() calls a program verifier to prove:
  The final SymT $\Rightarrow \varphi$
  
  • Charlie.com:#finish(Bob.com::#compare(Alice.com::#grab())) $\Rightarrow \varphi$ , the execution is accepted. 
  
  • Charlie.com:#finish(Alice.com::#grab()) $\Rightarrow \varphi$ , the execution is rejected.

• Note that the program verification is symbolic (only about code). The concrete values are ignored.
  • A middle ground between offline symbolic verification and runtime concrete checking.

• SVX’s performance overhead is near-zero
  • Because the theorems can be cached.
  • All normal executions should hit the cache.
Theorem cache and verification server

(SymT → ϕ)?

cache hit
(0 ms)

cache miss
(30 ~ 50 secs)

Theorem cache

Charlie.com

De-hash the hash values

Synthesize a straight-line program to be verified

C# program verifier

SVX verification server on the cloud
Our open-source project: SVAuth

Safer SSO integration solutions based on SVX
The SVAuth framework: SVX with OO

- Defines “login safety” and “login intent” properties at the base class level.
- Every concrete implementations are guaranteed to satisfy the base class level properties!

**Diagram:**

- **Generic Auth**
  - **OAuth 2.0**
  - **OpenID 2.0**
  - **OpenID Connect 1.0**
- **Protocol-independent level (defining safety properties)**
- **Protocol level**
  - **Facebook Connect**
  - **Google Login**
  - **Microsoft Live ID**
  - **Yahoo Login**
- **SDK level**
  - **Website-specific customizations**
  - **a.com**
  - **b.com**
  - **c.com**
  - **d.com**
  - **e.org**
A decades-old problem in verification

- Liskov Substitution Principle (LSP) tries to ensure that
  - If a property is true for the base class, then it holds for all derived classes.

```c++
class Rectangle {
    int height, width;
    virtual int getHeight() {return height;}  // LSP says to override.
    virtual int getWidth() {return width;}    // LSP says to override.
    virtual void SetHeight(int x) {height=x;}
    virtual void SetWidth(int x) {width=x;}
}

void foo(Rectangle r) {
    int w=r.getWidth();
    r.SetHeight(3);
    Assert(w==r.getHeight());
}

class Square: Rectangle {
    override void SetHeight(int x) {
        height=x;
        width=x;
    }
    override void SetWidth(int x) {
        height=x;
        width=x;
    }
}

Rectangle r = new Rectangle();
Assert(foo(r));

Rectangle r = new Square();
Assert(foo(r));
```

For SVX, there is not confusion
How does SVAuth work?

- SVAuth consists of an **agent** and an **adapter**
  - Agent: public agent, organizational agent or localhost agent
  - Website developer picks an agent, and sets its endpoint in the SVAuth config file
  - Copy the adapter folder onto the website
## Why adopting SVAuth?

<table>
<thead>
<tr>
<th></th>
<th>Traditional Oauth SDKs</th>
<th>SVAuth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oauth protocol</td>
<td>X Need some understanding of Oauth protocols</td>
<td>V Don’t need to know anything about SSO protocols.</td>
</tr>
<tr>
<td>App registration</td>
<td>X Register an app for each identity providers</td>
<td>V Don’t need to register apps for each identity provider</td>
</tr>
<tr>
<td>App secret</td>
<td>X Manage app secrets</td>
<td>V Don’t have to manage app secrets</td>
</tr>
<tr>
<td>Oauth SDK</td>
<td>X Import and update Oauth SDKs for each language</td>
<td>V Use only basic cryptographic primitives, no external dependencies</td>
</tr>
<tr>
<td>Security issues</td>
<td>X Do you trust Oauth SDKs and Oauth framework?</td>
<td>V Organizations can run their own version of SVAuth agent</td>
</tr>
</tbody>
</table>
SVAuth demo
Adopting SVAuth on your website -- extremely simple

Start login flow by redirecting to public agent
“http://authjs.azure.com:3020/login/Facebook”

Listen for the user’s identity information on
“/SVAuth/adapters/py/RemoteCreateNewSession.py”

Start the session
Our experience

• Current status
  • Support 7 SSO services and 3 languages (ASP.NET, PHP and Python)
  • Will support more.

• Integration with real-world applications
  • MediaWiki (8 lines of code changes)
    • Used by a Microsoft Research internal website.
  • HotCRP (21 lines of code changes)
  • CMT (10 lines of code changes)

• Open source, available on GitHub
  • Project repo: https://github.com/cs0317/SVAuth
  • Sample code: https://gist.github.com/pmcao/22d1c6f04ebd662c4baf83d7a6d1e9dd
  • Live demo: http://svauth-python-adapter.herokuapp.com/
• Most website programmers are not experienced “locksmiths”
  • Installing an SSO lock securely on a website is not easy.
  • SSO security bugs are pervasive. Even big companies make mistakes.
  • The problem is well known in the security community.

• Self-verifying execution (SVX)
  • It is a “locksmith” built into a lock product.
  • The locksmith watches how the lock is opened, and asserts if it is logically sound.

• SVAuth – Open-source SSO framework based on SVX
  • Please adopt SVAuth on your websites
  • Or, join the project to improve the code.
  • Let’s fundamentally address the SSO security bugs.