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 guide, drill holes, and install a lock cylinder, knobs and steel 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:
  • An attacker can sign into a victim’s account
  • An attacker can stealthily cause the victim to sign into the attacker’s account
    (commonly known as login request forgery)

• 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
  • Attacker logs into any victim’s account
  • Video

• Demo 2:
  • https://web.skype.com
  • Login 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

Google ID service

Relying party website

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

Redirection 1:
- **Realm:** the RP’s domain
- **Required:** (email, firstname, lastname)

Google ID service

Redirection 2:
- **Signed:** (firstname, lastname)
- **Signature:** “HRU436ETQ95TR939”
- **Firstname:** Bob
- **Lastname:** Johnson

Bob’s browser

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

Relying party website

Redirection 1:
- **Realm:** the RP’s domain
- **Required:** (email, firstname, lastname)
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 (纨绔) is the President!

Check out my photo with this token.

8
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.
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 satisfy $\varphi$”.
Example: comparing integer constants among three websites

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

**Alice.com**

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

**Bob.com**

```c
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**

```c
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

Client

<arbitrary, “nobody”>

<10, “Alice”>

<10, “Alice”>

<40, “Bob”>

<40, “Bob”>

Alice.com (10)

Bob.com (40)

grab

compare

Charlie.com (5)

<40, “Bob”>

conclude

finish
The system is vulnerable!

client

<arbitrary, "nobody">

<10, "Alice">

<Alice.com (10) grab finish conclude>

<10, "Alice">

<10, "Alice">

<10, "Alice">

<Client.com (10) grab finish conclude>

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<10, "Alice">

<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:

\[ \text{The final SymT} \implies \varphi \]

• `Charlie.com:#finish(Bob.com::#compare(Alice.com::#grab())) \implies \varphi\] , the execution is accepted.

• `Charlie.com:#finish(Alice.com::#grab()) \implies \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

(charlie.com)

SVX verification server on the cloud

C# program verifier

Recover the code of executed methods

Synthesize a straight-line program using SymT

Theorem cache

(\text{SymT} \Rightarrow \phi) ?

cache hit

(\equiv 0 \text{ ms})

cache miss

(30 \sim 50 \text{ secs})

Recover the code of executed methods

Synthesize a straight-line program using SymT

C# program verifier
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!
Liskov Substitution Principle (LSP) tries to ensure that:

- If a property is true for the base class, then it holds for all derived classes.

```csharp
class Rectangle {
    int height, width;
    virtual int GetHeight() {return height;}
    virtual int GetWidth() {return width;}
    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.GetWidth());
}

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.
Adopting SVAuth on your website -- extremely simple

- 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

- Assuming website *foo.com* is in PHP, and wants to do Facebook SSO
  - Simply redirect to 
  - Magically, the user’s identity information is available in these session variables
    
    ```
    Session['SVAuth_UserId'] = 108376550318508459185
    Session['SVAuth_FullName'] = John Doe
    Session['SVAuth_Email'] = johndoe@gmail.com
    Session['SVAuth_Authority'] = Google.com
    ```
  - Website programmers don’t need to know anything about SSO protocols.
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 repository: https://github.com/cs0317/SVAuth
SVAuth demo
Demos

• Buggy code
  • Remove cache entries
  • Comment out the line `stateGenerator.Verify` in `Facebook.cs`
  • *Login Intent* won’t pass.

• Correct code, first execution
  • Program verification is triggered
  • Both *Login Safety* and *Login Intent* pass the verification.

• Correct code, second execution
  • Theorems hit the cache, near-zero runtime overhead
• 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.