Usable Online Privacy

Browser-Based End-to-End Encryption Delivered by a Service Operator

Joshua Korner-Godsiff - u4685222
Supervisor: Roger Clarke

Australian National University

19th May 2014

Table Of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

The Problem

The NSA is spying on everything everyone does online.

The Problem

So are the intelligence agencies of:

UK, Canada, NZ, Australia, Austria, India, Netherlands, Russia, China, Iran, Syria, Israel

And probably many more.

The Problem

Seeing more serious and sophisticated security breaches (“hacks”) of large organisations.

Things like Heartbleed don’t help.

The Problem

Some organisations are just giving their users' data away.

(For money).

The Problem

How is the average, everyday user supposed to protect their privacy?

How is the average company supposed to protect itself and its users?

The Solution?

The Solution?

- Nice idea, but average user does not know
  - What encryption is.
  - How to set it up.
  - How to use it.
- Doesn't care.
- PGP invented in 1991. Virtually unknown outside tech circles.
- HTTPS, on the other hand...

The Solution?

Need a way to provide encryption **WITHOUT** impacting the user experience.

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. **Problem Definition**
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

Goal

Protect all users, not just those who seek protection.

Only the user, and any other users they give permission to, get access to that user's sensitive data.

Threat Model

Large-scale breaches of privacy result from compromise of a server/service/service-operator, not user machines.

Want to protect against **ANY** compromise on the server end, regardless of how or why that compromise occurs.

Threat Model

Sensitive data must be safe, even if the service operator itself is actively working against its users.

By extension:

- Includes the possibility that the server/service is under the control of hackers, intelligence agencies, or any other adversary.
- Any threats between the service and client (i.e. threats to communication).

Threat Model

Not appearing in this threat model:

- Client-side threats
- Other users

Usability Requirements

Solution must avoid altering user experience, or erecting barriers to uptake.

- No user-initiated downloads.
- No installation.
- No configuration.

$\Longrightarrow$ Must be as mass-deployable as the service itself.

Usability Requirements

Your grandma has to be able to use this.

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

Solution Overview

- Encrypt all sensitive data on client.
- Use public/private key-pair to send data to other users.
- Use password to encrypt all encryption keys.

Solution Overview

- Store encrypted information and encrypted keys with the service.
- **DO NOT** store unencrypted information, or unencrypted keys, or password with the service.

Solution Overview

Have service deliver encryption capabilities to the browser, using JavaScript.

Problems To Be Solved

Encryption via service-delivered JavaScript means service can modify encryption code, at any time, without notice.

Some third party adversary could also change it.

Prefer this did not happen, or that we could detect it.

Problems To Be Solved

Need a way to check:

1. Code the client receives is what the service intended to send.
2. Cryptographic & privacy preserving functions are what user expects to receive.
    - Need to standardise!
3. The rest of the code isn't violating the user's privacy, nor subverting security measures.

Problems To Be Solved

Can't rely on service to provide method to check.

Can't rely on all users installing something to check.

Problems To Be Solved

CAN (statistically speaking) rely on *some* users installing something to check.

Those users can tell others.

Knowing is half the battle, as not using the service unless/until the problem is fixed will keep data safe.

Problems To Be Solved

How do we actually perform these checks?

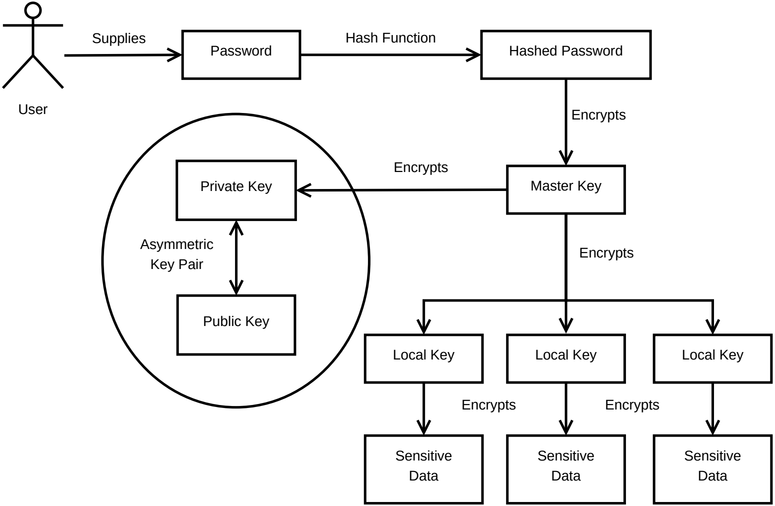
Problems To Be Solved

How can users exchange data securely, if the service is facilitating the exchange?

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. **Interesting Bits**
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

Encryption Scheme



Storing Sensitive Information

For sensitive information present on the page at load time:

```
<div class="sensitive" key="encrypted-local-key-here">
    Encrypted information here.
</div>
```

Decrypt on page load.

DON'T decrypt anything not marked as “sensitive”.

Doesn't have to be a div.

Storing Sensitive Information

For information loaded in after page load, have to ensure

- It is tagged as sensitive.
- Has encrypted key attached

when it is added to DOM.

Checks

Use browser plug-in.

- Can hook into web-pages, JavaScript files, DOM.
- Can hook into JavaScript runtime, if need-be.
- Alerts users if privacy may be violated. Stops page running until user chooses what to do.

Check 1

“Code the client receives is what the service intended to send”.

- Use HTTPS.
- Have the service sign anything it sends with private key.
- Store public key in certificate.
- Protects against everything except purposeful compromise by the service.
    - Other two checks negate that danger.

Check 2

“Cryptographic & privacy preserving functions are what user expects to receive”.

- Roll all these functions into a single JS file. Hash it.
- Store this hash in the plugin itself.
- Check for it on page load.

Check 3

"The rest of the code isn't violating the user's privacy, nor subverting security measures".

The hard one! Many different aspects to this.

Use plugin to parse all JS present at page load.

And then...

Check 3

"The rest of the code isn't violating the user's privacy, nor subverting security measures".

Check for

- eval, setTimeout, setInterval
  - Allows arbitrary code execution.
- Inline JavaScript
  - Unpredictable.
  - Don't need it.
- Loading of *any* novel external resources (images, CSS, JS files, PDFs).
  - Can be used to convey information to the server.
  - All resources need to be loaded at page load.
- Messing with the crypto-code.

Check 3

"The rest of the code isn't violating the user's privacy, nor subverting security measures".

Perform Taint Analysis on the JavaScript code to detect potential privacy violations.

Just what the heck is Taint Analysis?

Taint analysis is a analysis technique.

- Comes in Static and Dynamic flavours.

Usually used to detect if unsanitised input is ending up somewhere dangerous. For example:

- User input being fed directly to SQL queries.
- User input being fed directly to exec command.

We're going to use it to see if sensitive data is being sent anywhere (e.g via Ajax call).

Just what the heck is Taint Analysis?

To perform taint analysis, you need:

- Software source-code. (Byte-code may suffice).
- Set of tainted data sources. (E.g. User input).
- Set of ‘taint-sink’ functions. (Where you don’t want tainted data to go).
- Taint propagation model for the programming language(s) the software is written in.

Taint Analysis for JS

- Source-Code ✔
- Tainted data-sources
  - Any function in the DOM API that could pull decrypted data from the DOM.
  - Decryption functions.
  - Key retrieval functions.
- Taint-sink functions
  - XML HTTP Requests (a.k.a. Ajax calls).
  - Anything else that can be used to communicate.

Taint Propagation Model

Expressions & Function Calls

Won't go into complete detail. (Long).

Expressions:

- If any sub-expression is tainted, whole expression is tainted.

```
42 * foo + bar;
```

Function Calls:

- Any arg tainted, whole call tainted.

```
func(arg1, arg2, ...);
```

Taint Propagation Model

Assignments

- Expression tainted $\Longrightarrow$ identifier tainted

```
identifier = expression;
```

  - Can remove taint by assigning untainted expression.

- Propagates up objects/arrays, so:

```
foo[i] = x;
foo.bar = x;
```

Taint Propagation Model

References & Tainted Scopes

References

- Need to track.

```
var a = {foo: "Hello"}
var b = a;
b.foo = taintedEntity;
```

Tainted Scopes:

- If a block level construct (if, for, while, switch, etc) depends on tainted entity, whole scope is tainted.
- Taint persists after block.

Taint Propagation Model

If Statements

```
if (conditionalExpression) {
      statementA;
      ...
} else {
      statementB;
      ...
}
```

- If `conditionalExpression` tainted, both branches tainted.
- If not tainted, analyse branches independently.
- Principles apply to other conditionals (e.g. Switch).

Taint Propagation Model

While Statements

```
while (condition) {
        statements;
        ...
}
```

- If `condition` is tainted, all statements in loop body are tainted.
- Principle applies to all other loop statements.

Taint Propagation Model

Loop Statements in General

- Need to run taint analysis over loop repeatedly.
- Stop once taint status has stabilised.

```
var foo = untainted-entity;
var bar = untainted-entity;
while(untainted-condition) {
    var baz = bar;
    bar = foo;
    foo = tainted-entity;
}
```

- Don't allow untainting in loop body. (Infinite loops).

Taint Propagation Model

With Statements

- Disallow use of `with` statements under our model.
- Too difficult to reason about.

```
var obj1 = {b: "Hello"};
var obj2 = {foo: "World"};
(function (a, b) {
    with (a) {
        b = "Goodbye";
    }
})(obj1,obj2);
alert(obj1.b); //alerts "Goodbye"
```

Taint Propagation Model

Functions

Functions are complicated. Have to deal with:

- Reference parameters (i.e. pointers) being tainted.
  - Hard, because JS is untyped.
  - Use object/array notation (a[b] and a.b) to detect.
- Parameters tainting each other.
  - Assume each parameter is tainted (one at a time)
  - See what else becomes tainted.
- Out-of-scope variables.
  - Treat them as an implicit parameter.
- Return value tainted $\implies$ function itself tainted.

Taint Propagation Model

Functions

Functions get even more complicated.

Because functions are also objects. And objects are functions.

Taint Propagation Model

Objects

- If one member of an object is tainted, the whole thing is tainted.
  - Becomes way too complicated otherwise.
- In other respects, treat like functions.

Taint Propagation Model

Try-Catch & Exceptions

- Try-Catch-Finally blocks, and Exceptions, are not covered under model.
- Probably could be, but 'catch-all' nature of `catch` block makes this complex.

Making it Usable

The above is too restrictive, by itself.

- Wrap tainted functions to check if reading sensitive data.
- Don't return sensitive data from these functions.
  - Behave normally for regular data.
- Unwrapped functions still available. (Legitimate uses).

Data Laundering

Tainted values could be written to DOM, and read back as untainted values.

- Wrap DOM-write functions to add “sensitive” flags, which can be detected by wrapped DOM-read functions.
- Alert user if tainted value is written via unwrapped function.

Exchanging Sensitive Information

To send something to another user:

- Use public key crypto.
- Need a third-party CA (at least in the plugin), to protect against Man-in-the-Middle attacks from the service.
- Could also use web of trust.

No solution is perfect.

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. Conclusion

Impact & Future Work

- Targeted Advertising doesn't really work.
  - Some for most business models depending on sensitive user data.
- Sets intelligence agencies back to specific, targeted surveillance.

Future Work

- Taint-Analysis produces a lot of false positives.
  - Could be much narrower.
  - Dynamic tainting worth a look.
- Needs to be gone over with a fine-tooth-comb.

Table of Contents

1. Introduction
   - The Problem
   - The Solution?
2. Problem Definition
   - Goal
   - Threat Model
   - Usability
3. Solution Overview
   - Problems to be Solved
4. Interesting Bits
   - Encryption Scheme
   - Checks
   - Taint Analysis
   - Making The Model Usable
5. Impact
6. **Conclusion**

Conclusion

That was a gross over-simplification of my thesis.

Any questions?