# Secure web browsing with the OP web browser

Paper by Chris Grier, Shuo Tang, Samuel T. King

Presented by Farhan Jiva

# Overview

1. Motivation
2. The OP browser design and implementation
3. Security policy and enforcement
4. Formal verification
5. Analyzing browser-based attacks
6. Evaluation

# 1. Motivation

- Current web browsers are fairly vulnerable
  - All major browsers affected (IE, FF, Safari, Opera)
- Authors claim the reason is because of a flawed design and architecture
- Proposing a new design for a browser
  - Relies on operating system principles

# 2. The OP browser design and implementation
## Threat model and assumptions

- Attacks from a web page can target any part of the browser
- An attack can be any sort of form
  - Namely, code injection/execution
- Assumed that the OS and JVM will isolate the browser's subsystems
- Assumed that DNS names are correct

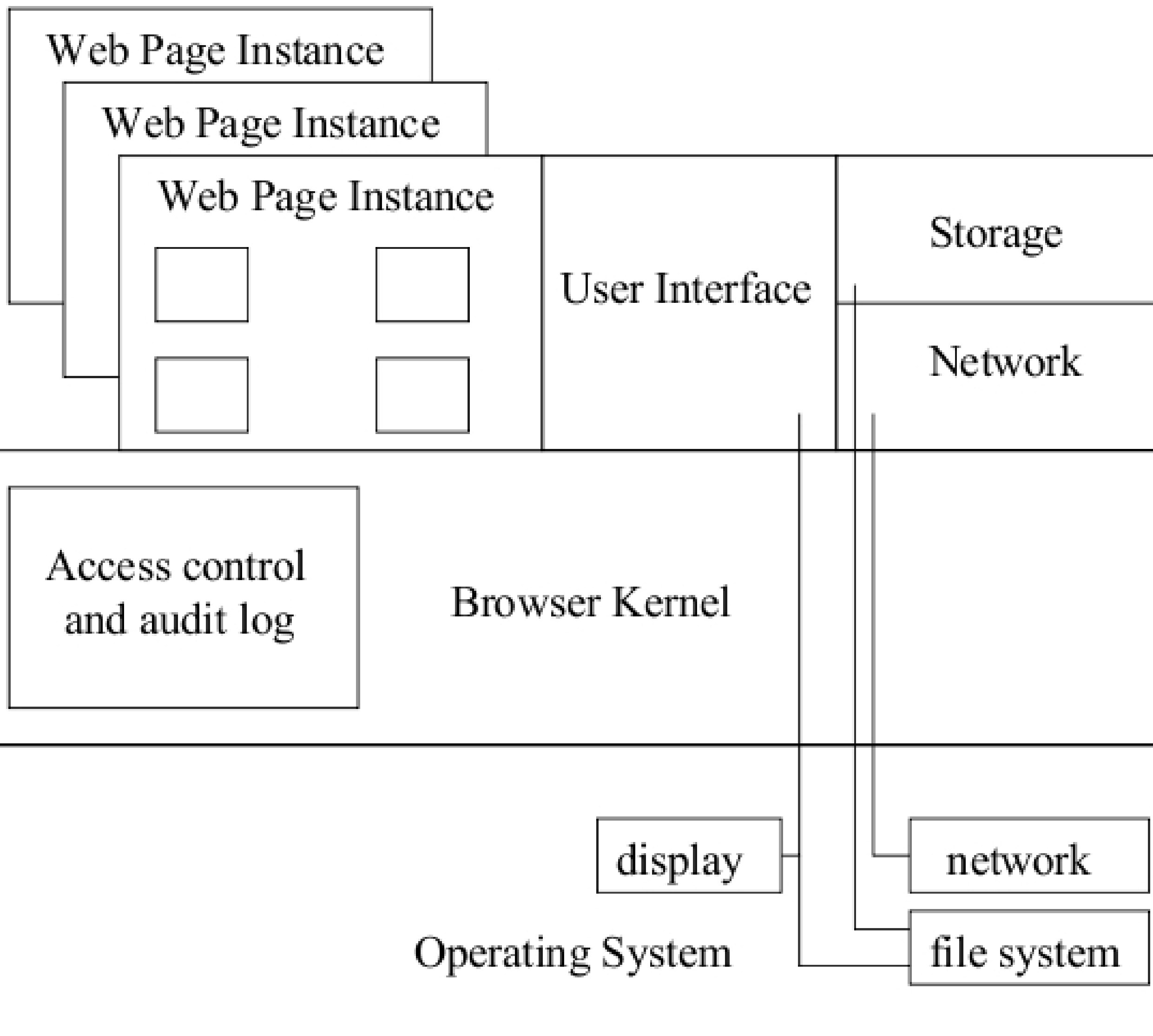
# 2. The OP browser design and implementation

## Design principles

- Simple and explicit communication between the parts of the browser
- Strong isolation between these components
- Design the components properly
- Maintain compatibility with current technologies

# 2. The OP browser design and implementation

## OP browser architecture



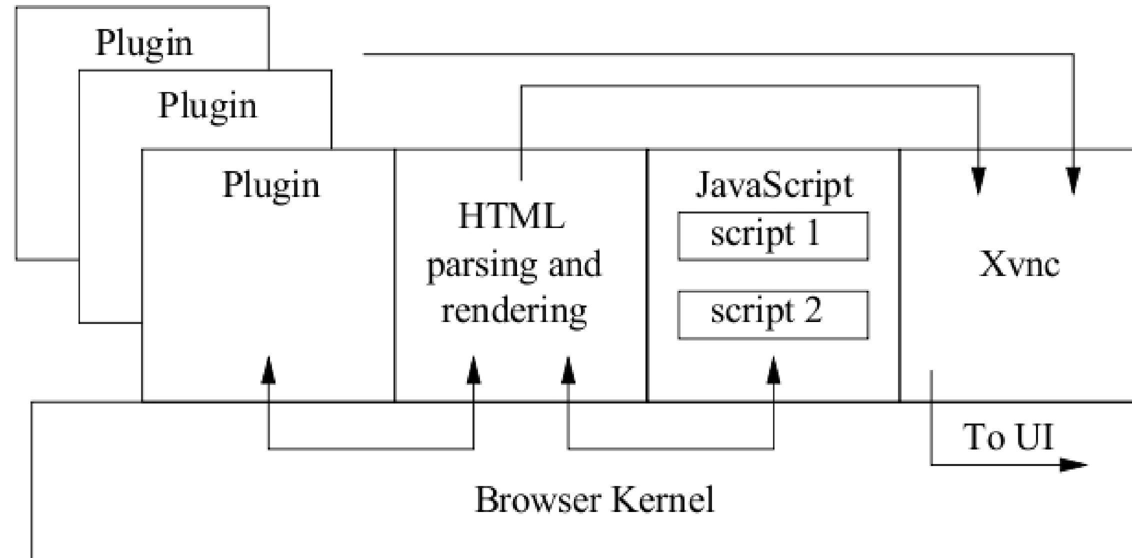
(a) Overall architecture of our OP web browser

- 5 main subsystems
  - Web page
  - Network
  - Storage
  - User Interface
  - Kernel
- Each run within separate OS-level processes
- Each part communicates through kernel
- Each part sandboxed using SELinux

# 2. The OP browser design and implementation
## The browser kernel

- 3 main roles
  - Manage subsystems
    - Creating/destroying processes
  - Manage messages between subsystems
    - OS-level pipes
  - Maintain detailed audit log
    - Assists with forensic analysis in case of compromise
- Single-threaded, event-driven

# 2. The OP browser design and implementation
## The web page subsystem




(b) Overall architecture of a web page instance

- Each web page instance is an individual web page
- Each web page instance contains a set of OS-level processes
  - HTML engine
    - KHTML (written in C++)
  - JavaScript engine
    - Rhino (written in Java)
  - Plugin instances
  - X-server

# 2. The OP browser design and implementation
## User interface, network, storage subsystem

- UI subsystem
  - Written in Java
  - Full access to the file system
  - Includes address bar, navigation buttons, menus, etc.
- Storage subsystem
  - Stores cookies and such in an Sqlite database
- Network subsystem
  - Implements HTTP

# 3. Security policy and enforcement
# Browser plugins

- Plugins let you view additional content in your web browser
  - Flash player, PDF viewer
  - Plugin determined by MIME-type
- Plugins complicate browser security
  - Run in the same address space as the browser
- Currently, plugin writers implement their own ad-hoc security features

# 3. Security policy and enforcement
# Plugin security in OP

- Each plugin must pass its messages through the browser kernel
- Each plugin is run in a separate process
  - Each process is labeled with a security context (domain name)
- This label is used to make decisions for plugin and browser actions
  - A plugin can be denied access to a browser resource
  - The rest of the browser can be denied access to a plugin resource

# 3. Security policy and enforcement
# Plugin security policies

1. Provider domain policy
   - Sets the origin of the plugin to the site hosting the plugin content
   - The plug can then access cookies, make network connections to its corresponding host
2. Plugin freedom policy
   - Some plugins need more flexibility
     - Example: a peer-to-peer video chat plugin
   - Solution: allow the plugin to access storage and network subsystem

# 4. Formal verification

- The authors decided to formally verify the correctness of OP
- They use a modeling interpreter/language called Maude
- Using Maude, they find invariants
  - Program invariants
    - Can be easily gathered from the source code
  - Visual invariants
    - E.g. preventing address bar spoofing

# 4. Formal verification
# Modeling using Maude

```
1 mod SIMPLE-CLOCK is
2   protecting INT .
3   sort Clock .
4   op clock : Int -> Clock [ctor] .
5   var T : Int .
6   rl clock(T) => clock((T+1) rem 24) .
7 endm
```

Fig. 3. A simple Maude example from the Maude Manual (Version 2.3). This example describes a model for a 24 hour clock in Maude.

```
search in SIMPLE-CLOCK :
clock(0) =>* clock(T)
such that T < 0 or T >= 24 .
```

Fig. 4. The search statement from the Maude Manual (Version 2.3) showing how to model check the SIMPLE-CLOCK model invariant using Maude's search functionality.

- *sort* is similar to the class keyword in C++, defines a category
- Fig. 4 shows how to use Maude to find internal states which violate an invariant (i.e. if the 24 hour clock holds an illegal value)

# 4. Formal verification

## Formal models and system implementations

- Often, there is a gap between a formal model and its corresponding implementation
- The authors believe their implementation is very similar to the formal model
- The formal model they created is focused on message passing between the components

# 4. Formal verification
# Modeling the OP browser

```
<UI-ID : Frame | addrBar: URL, ... >
imsg(count, src, dst, IDENTIFIER, content)
< ... > ...
```

Fig. 5. The message specification in Maude. The first section of the specification is a class-like structure, starting with $<$ and ending in $>$. UI-ID is the instance identifier of the type, Frame is the type, and after the pipe are the members of the type. The next line begins with imsg and is the constructor for the message type. The constructor takes the elements in parenthesis and creates an object of a specific type. The imsg constructor creates an object of type Message.

- Messages are tagged with a count to ensure in-order processing
- Message ordering is preserved by the browser kernel

# 4. Formal verification
# Modeling the OP browser

- Modeling user actions
  - Maude rule below describes the message generation as a result of a user clicking the “GO” button

```
< UI-ID : Frame | addrBar: URL, ... > GO
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : M >
=>
< UI-ID : Frame | addrBar : URL, ... >
imsg(M, UI-ID, KERNEL-ID, MSG-NEW-URL, URL)
< MSG-ID : MsgCount | msg-to-process : N, msg-to-send : s(M) >
```

Fig. 7. Maude expression for the “GO” UI button causing a message to be sent. The first line represents the portion of the browser state for the Frame and the user action being performed, which in turn causes produces a new Frame state and the message with type set to MSG-NEW-URL.

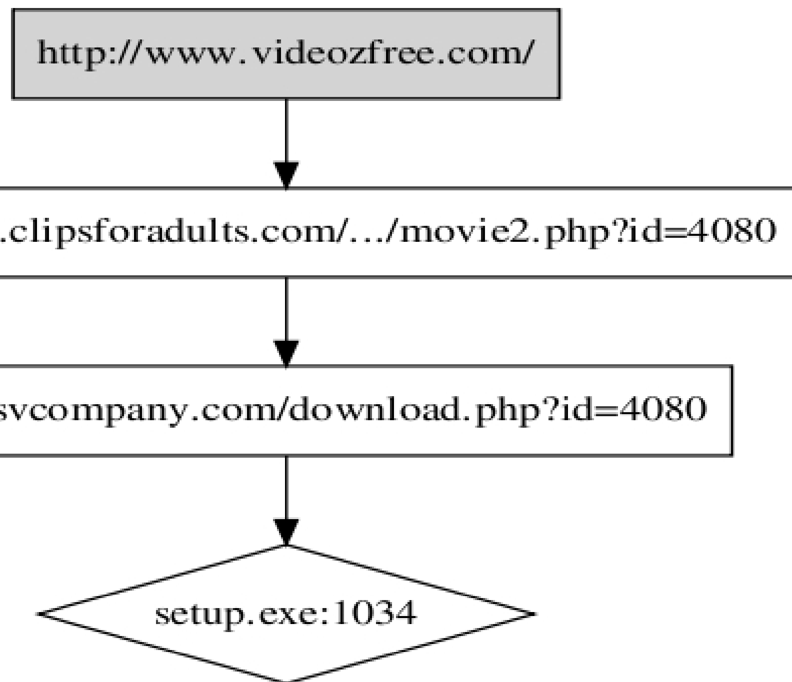
# 4. Formal verification
## Modeling the OP browser

- Model checking address bar invariants
  - Start with defining the “good” browser states
  - Then use Maude to search for “bad” ones

```
< UI-ID : Frame | AddrBar : S1:String, NavWebApp : WebApp1:Int , ... >
< WebApp2:Int : WebApp | Content : S2:String, ... >
such that (WebApp1:Int == WebApp2:Int) /\ (S1:String =/= S2:String)
```

Fig. 8. A Maude expression describing the condition checked for address bar spoofing. This condition is used as a test for bad browser states. The first line is the current state of the browser, specifying the UI and ID for an instance of the web page subsystem. The last line is the comparison, which checks that the URLs associated with the address bar and web page subsystem are different, indicating a state where the address bar is spoofed.

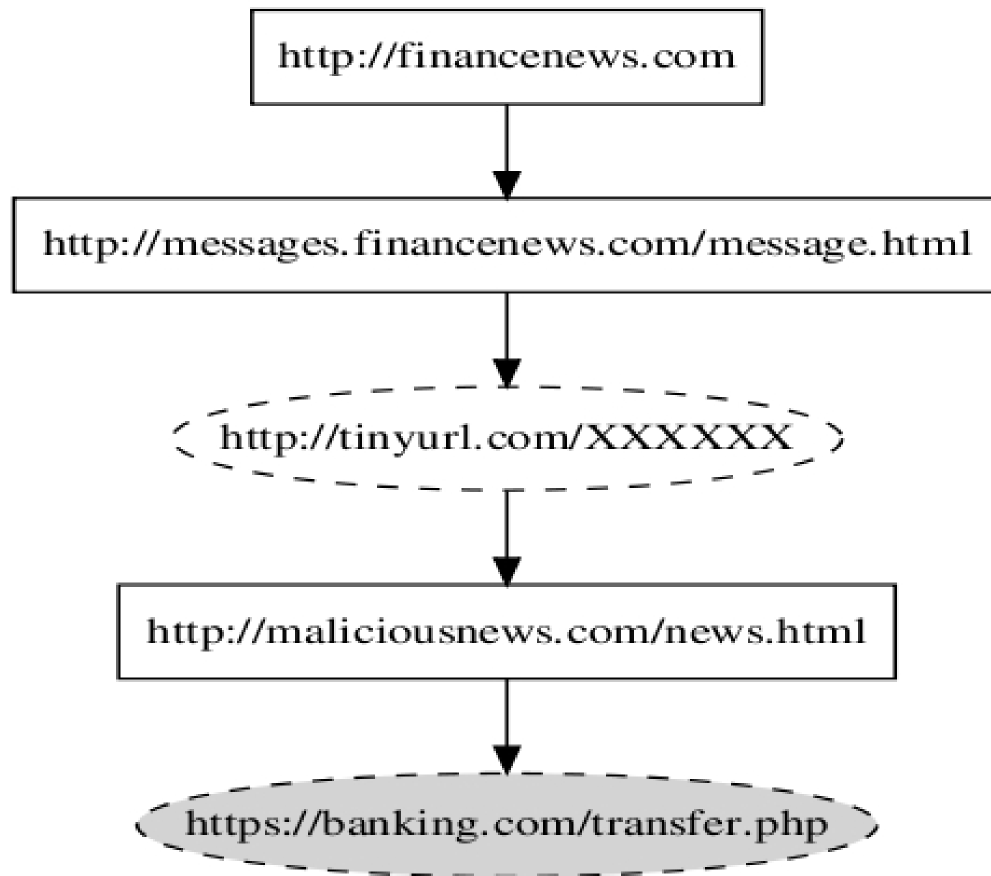
# 5. Analyzing browser-based attacks
# Intrusion analysis design




(a) Forward dependency graph for *videozfree* attack

- Even though they secured their browser, some attacks can still occur
  - Namely, social engineering attacks
- Solution, track web pages and files and the events which connect these together
  - Also, make a pretty graph out of this information

# 5. Analyzing browser-based attacks
# Example: cross-site request forgery




(b) Backward dependency graph for *xsrf* attack

- Uses the BackTracker graph generation algorithm to generate dependency graphs
- CSRF steps
  1. Authenticate yourself with your bank website
  2. Visit a malicious website
  3. The malicious website uses your authenticated browser to transfer funds out of your bank account

# 6. Evaluation
## Performance evaluation

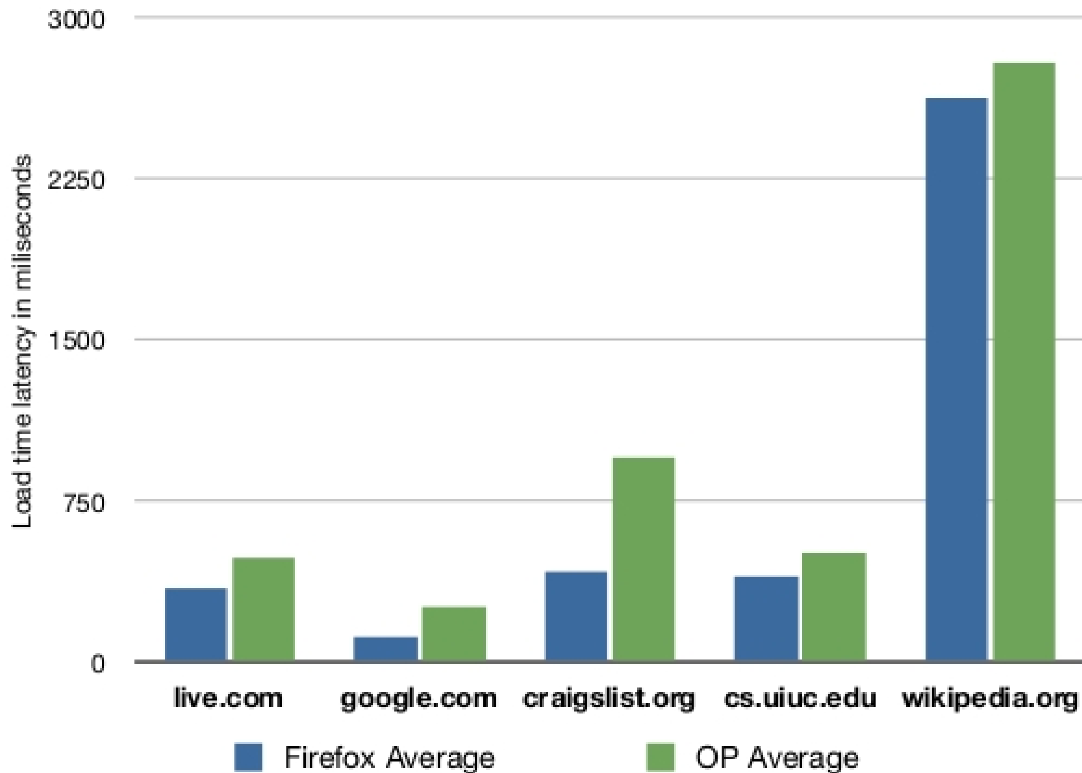



Fig. 11. Loading latencies for OP and Firefox.

- Used a 2.66GHz Intel Core 2 Duo machine with 2GB of RAM and 250GB SATA
- Running 64-bit Fedora 7
- Compared OP against Firefox 2.0.0.12
- Loaded each web page 5 times and took average
  - No caching
- Authors claim the latency times are due to the JavaScript engine

# 6. Evaluation
## Security analysis

- Authors understand that their browser could still be compromised
- Web page compromised?
  - Hijack message sending to other subsystems
- UI, storage, network compromised?
  - Tweak the UI (address bar spoofed)
  - File system access, access to persistent storage
  - Arbitrary network connections
- Browser kernel?
  - Full browser compromise
  - But don't worry, it's only written in 1221 lines of C++

# Questions?