KU Leuven DistriNet Research Group

Building on sand: Secure software on insecure platforms?

Frank Piessens

K.U.Leuven, Belgium

ACKNOWLEDGMENT: These slides include material from slide decks created by other DistriNet researchers, including Pieter Agten, Willem De Groef, Lieven Desmet, Dominique Devriese, and Raoul Strackx.

Overview

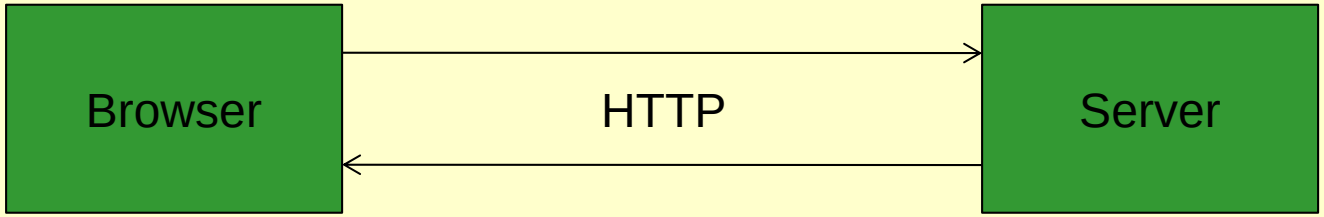
- Introduction
  - Some key challenges for software security
- Secure compilation to native code
- Secure browsers
- Conclusions

We expect too much of developers!

- Understanding whether a piece of C code is secure requires:
  - Understanding of the C language
    - Approx complexity: 700 pages of spec
  - Understanding the details of the compiler
    - Approx complexity: 3.7 million lines of code
  - Understanding the runtime library implementations
    - Approx complexity: 1.7 million lines of code
  - Understanding the operating system
    - Thousands of pages of specs and millions of lines of code
  - Understanding the details of the processor and other hardware

And the Web is many times worse!

- It looks deceptively simple from a distance:



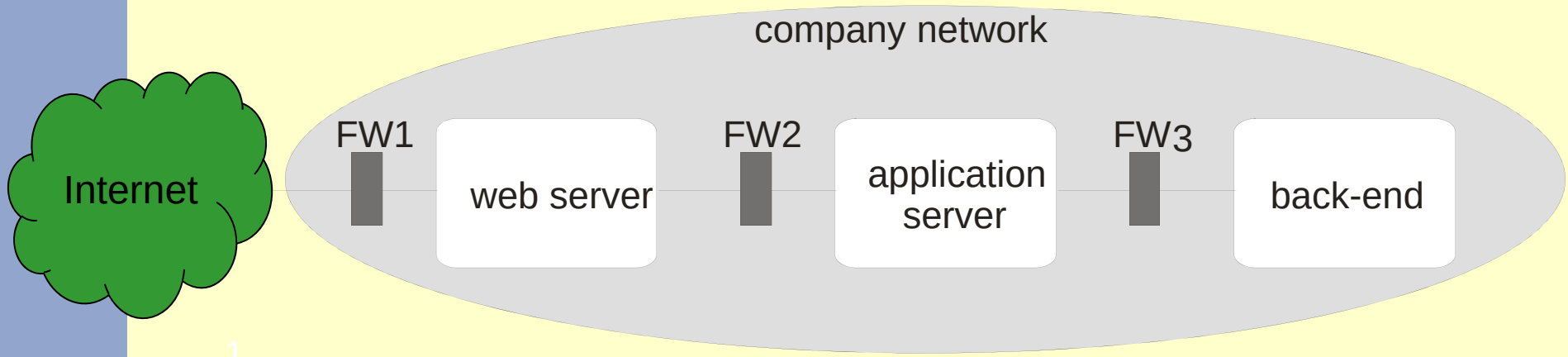
- But each of these components is staggeringly complex
- And they interact in unforeseen ways
- Let's look at each of them in turn

The Browser

- Displays HTML
  - The HTML5 spec is several hundreds of pages
- Executes JavaScript
  - The ECMAScript 5.1 spec is several hundreds of pages
- Supports plugins
  - Flash alone is as complex as JavaScript
- Supports a wide variety of protocols
  - http, https, ftp, file, telnet, mailto, gopher, ldap, …
- Supports a growing set of API's
  - Audio, video, geolocation, client-side storage, messaging, …
- Supports isolation between content from different sources
  - i.e. a browser is more or less an operating system

The Server

- Is typically an intricate distributed system itself:



1

Web and Application server :

- Static HTML
- Dynamic content generation: JSP, ASP, CGI, PHP, …
- J2EE, .NET, COM+

Back-end:

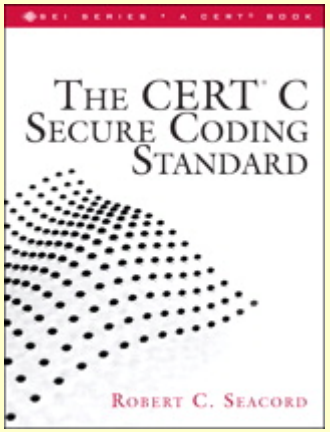
- SQL based DB
- Mainframe
- Directory server

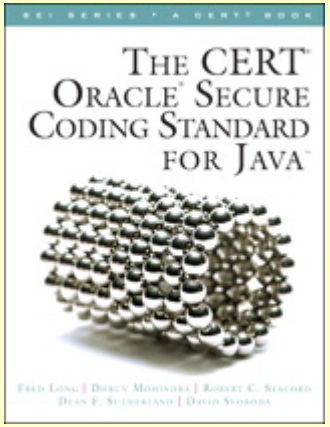
HTTP

- Stateless
  - But many mechanisms to add state on top
- “Simple” protocol methods, that do arbitrary complex things
- A proliferation of header fields
  - That each need their own standard to describe what they do
- Redirects
  - Turn a simple request in a distributed computation
- Relies on DNS
  - Cfr. DNS-changer virus in the news the past weeks
- And HTTP is only one of the many web-protocols!

How do we deal with this today?

- Coding guidelines and tooling
  - For instance: 89 Rules and 132 Recommendations in the CERT C Secure Coding Standard
  - Source code analysis tools implement heuristic checks to detect deviations from these rules
- Ad-hoc countermeasures in compiler / OS
  - Stack canaries / ASLR / taint-mode / …
- This can lead to substantial software security improvement
  - But is not the long-term solution





Two key challenges

- The programming language is supposed to isolate the programmer from details of the platform to which the code is compiled
  - This fails miserably as far as security is concerned
- The platform is supposed to provide basic security guarantees to applications
  - What is provided is a complete mismatch for what applications need today
- In this talk we will discuss some directions to rectify this situation

Overview

- Introduction
- Secure compilation to native code
  - What does it mean for a compiler to be “secure”?
    - The principle of “source-based reasoning”
  - How can we achieve secure compilation on commodity platforms?
- Secure browsers
- Conclusions

What is “secure” compilation?

- The compiler is the tool that is supposed to isolate the programmer from the low-level platform.
  - It succeeds well with respect to “expected functionality” of the code
  - It fails with respect to “security properties” of the code
- What are today’s compilers missing? What would make a compiler “secure”?

Security depends on the power of attacker

- Case 1: The attacker can only provide input to the program under attack
  - Example: a network service running on a hardened and well-protected server machine
  - For this case, a secure compiler should make sure that behavior of programs is *well-defined for all possible inputs*
- Case 2: The attacker can interact with the program at a lower-level
  - Example: any client machine (where malware is a realistic threat), or situations where the attacker can load code
  - For this case, a secure compiler should *preserve contextual equivalence*

Case 1: high-level attackers

- A programming language is *safe* if its behavior is always well-defined
  - E.g. a[i] = (int) x.f()
- Examples:
  - Safe languages: Java, C#, Scala, …
  - Unsafe languages: C, C++, Pascal, …
- A compiler is *safe* if any undefined behavior leads to immediate termination
  - Compilers for safe languages are always safe
  - Fully safe compilers for C typically have terrible performance

Case 1: high-level attackers

- A safe compiler
  - Protects its own abstractions (e.g. no stack smashing attack)
  - Is inherently portable
  - Mitigates the security impact of developer oversights/bugs!
- An unsafe compiler puts the burden of avoiding undefined situations on the programmer
- This is exactly why it is easier to write secure software in Java than in C
- But C compilers also get closer and closer to being safe

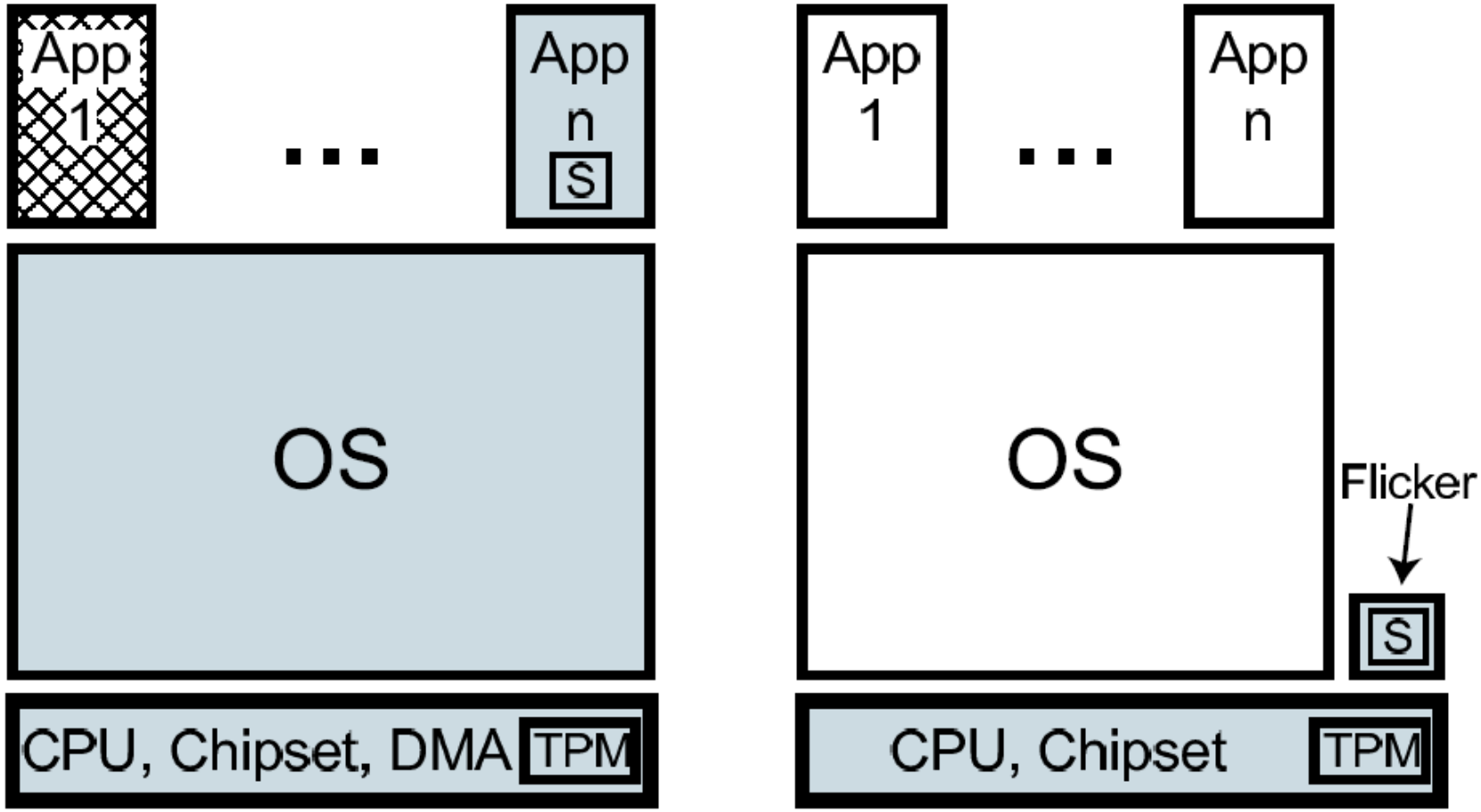
Case 2: low-level attackers

- In many cases, attackers can do more than just provide input, for instance:
  - Because they infected the OS with malware, or
  - Because the application supports plugins, or
  - Because the attacker can perform a code-injection attack against native code in the run time, or
  - …
- All current (state-of-practice) compilers give up any form of protection for this case
  - As a consequence, it is impossible for instance to do secure web-banking

Case 2: low-level attackers

- Can we compile “securely” against low-level attackers?
- Some recent breakthroughs make this possible!
  - A key enabler is the development of security architectures to support on-demand isolated code execution on commodity hardware
  - See for instance the PhD thesis of Bryan Parno, winner of the 2010 ACM Doctoral Dissertation Award

Isolated execution of critical code



(Picture taken from Parno's PhD thesis)

Secure compilation to native code

- To construct a secure compiler:
  - We start from a safe source-language
  - We develop a native-code security architecture using techniques similar to Parno's Flicker
  - We develop a compilation scheme from the source-language to the native-code security architecture
  - We show that for this compilation scheme, low-level attackers have no more power than high-level attackers.

(This is a substantial part of the PhD thesis's of Raoul Strackx and Pieter Agten)

Safe source language

- Small, object-based, single-threaded
- Public methods, private variables
- Branches, loops, local variables
- Indirect method calls
- No dynamic memory allocation
- **Safe**

```
object o {
  M<(Int, Int)->Unit> lstnr = null;
  Int value = 0;

  Unit setLstnr(M<(Int,Int)->Unit> l){
    lstnr = l;
    return unit;
  }

  Int getValue() {
    return value;
  }

  Unit setValue(Int v) {
    if (lstnr != null && value != v){
      lstnr(value, v);
    }
    value = v
    return unit;
  }
}
```

Contextual equivalence

High-level objects provide encapsulation

```
object o {
  Int value = 0;

  [...]

  Unit plusTwo() {
    value += 2;

    return unit;
  }
}
```

O_1

```
object o {
  Int value = 0;

  [...]

  Unit plusTwo() {
    value += 1;
    value += 1;
    return unit;
  }
}
```

O_2

$O_1 \simeq O_2$: No third *test object* O_T can differentiate O_1 from O_2

High-level attackers

- It is the responsibility of the programmer of a module to protect against high-level attackers
  - Such attackers take the form of arbitrary high-level code interacting with the object
  - This supports the *principle of source based reasoning* for security:
    - One can find and understand any vulnerability in the code by only looking at and understanding source code
- A good way of thinking about security properties of code is in terms of contextual equivalence

Example: integrity of a field

```
object o {
  Int zero = 0;

  Int m(M<ϵ->Unit> cb) {
    zero = 0;
    Unit x = cb();
    if (zero == 0)
      return 0;
    else return 1;
  }
}
```

O_1

```
object o {
  Int zero = 0;

  Int m(M<ϵ->Unit> cb) {
    zero = 0;
    Unit x = cb();

      return 0;

  }
}
```

O_2

$O_1 \simeq O_2$ is saying "The callback `cb()` cannot modify the `zero` field"

Example: an object-invariant

```
object o {
  Int min = 0;
  Int max = 0;

  [...]

  Int m() {
    if (min <= max) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

O_1

```
object o {
  Int min = 0;
  Int max = 0;

  [...]

  Int m() {

      return 0;

  }
}
```

O_2

$O_1 \simeq O_2$ is saying "The `min <= max` invariant cannot be violated"

Summary

- Attackers are represented as test objects
  - High level attackers are source code test objects
  - Low level attackers are machine code test objects
- Successful attacks against security properties of a module
  =
  Contextual non-equivalence of the module with another module that “checks the property”
- Secure compilation should preserve contextual equivalence:
  - If an attack exists at the low level
  - Then, a low-level attacker can distinguish the two low-level modules
  - Hence, a high-level attacker can distinguish the two high-level modules
  - Hence, an attack exists at the high level
  - Hence, the attack can be explained at source code level

The low-level platform

- Standard Intel x86 style platform
  - Processor with
    - Program Counter
    - Registers and a Stack Pointer
    - Status (flags) registers
  - 32-bit memory space mapping 32-bit addresses to 32-bit words
- Extended with a program-counter based memory access control model

Sample instructions

`movl` r_d r_s	Load word at address r_s into r_d
`movs` r_d r_s	Store word r_s at address r_d
`movi` r_d i	Load the constant value i into r_d
`add/sub` r_d r_s	Arithmetic (sets flags)
`cmp` r_1 r_2	Compare (sets flags)
`jmp/je/jl` r_i	Jumps
`call` r_i	Call (pushes return address on stack)
`ret`	Return from call (pops return address from stack)
`halt`	Stop execution with result in R0

Standard compilation does **not** preserve contextual equivalence

```
object o {
  Int value = 0;
  Int secret = 0;

  [...]

  Int getValue() {
    return value;
  }
}
```

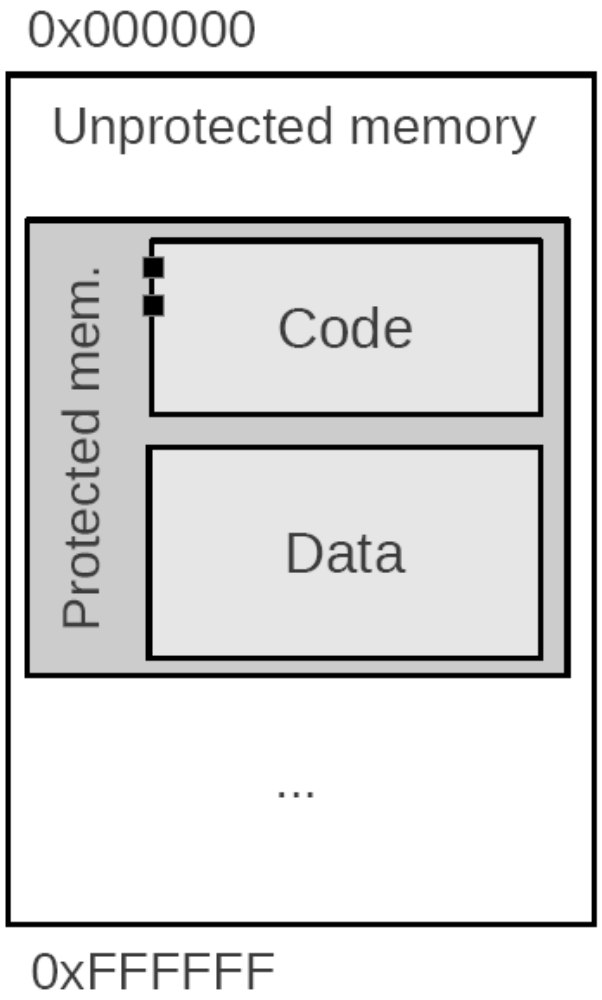
⇒

```
__getValue_:
  0x000000CC: movi R0 1
  0x000000CD: sub SP R0
  0x000000CE: movi R1 0
  0x000000CF: movs SP R1
              [...]

field0:
  0x60000001: data: 0
field1:
  0x60000002: data: 0
```

Low-level protection mechanism

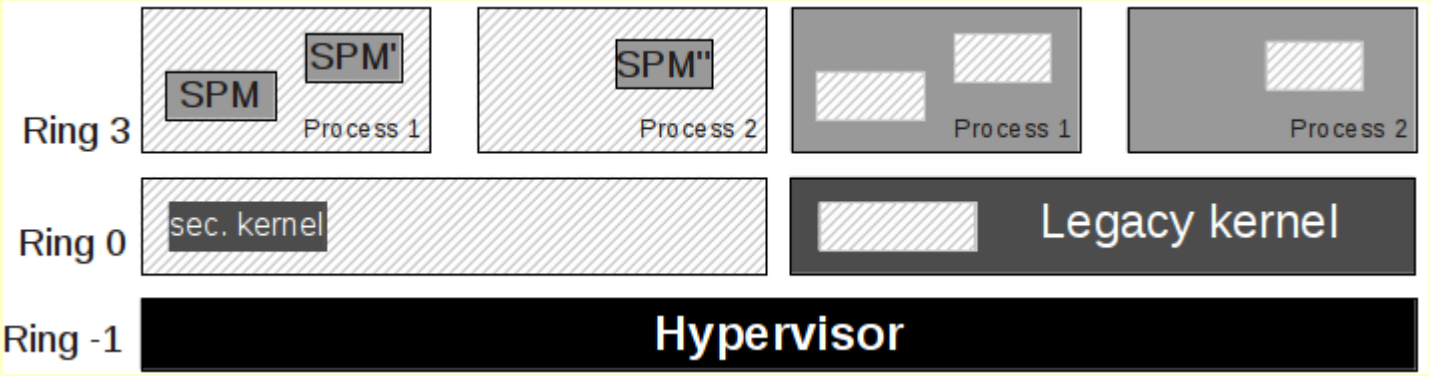
- Need some low-level protection mechanism
- Program counter-based memory access control

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x



Low-level protection mechanism

- This can be implemented efficiently!
- Two possible implementation strategies:
  - Flicker-style (has been implemented by Raoul Strackx)



  - In hardware (extend memory access control logic)

Compilation scheme

- As expected:
  - Compile methods and put in code section
  - Allocate space for fields in data section
  - Generate entry point for each method
  - …
- But many tricky details:
  - Handling returns of call-backs
  - Handling potentially “poisoned” function pointers
  - Protecting local variables / return addresses on the call stack
- Pieter Agten implemented a compiler and proved it secure
- Raoul Strackx implemented an efficient runtime platform to compile to

Secure compilation: conclusions

- We can securely compile one module, and provide very strong security assurance:
  - Against code injection attacks
  - Against malware (even kernel-level)
- But this is not a panacea
  - Source-level vulnerabilities remain the responsibility of the programmer
  - We still lack trusted user interface
  - It would be good to support multiple modules
    - (This actually works already in our prototype)

Example source-level vulnerability

```
object Acc {
 Int pin = 1234;
 Int count = 0;

 Unit test(Int t,
           M<Int->Unit> cb){
    if (count == 3)
       return unit;
    if (pin == t) {
      cb(0);
      count = 0; }
    else  {
      cb(-1);
      count = count+1; }
  }
}
```

Example source-level vulnerability

```
object Acc {
 Int pin = 1234;
 Int count = 0;

 Unit test(Int t,
           M<Int->Unit> cb){
    if (count == 3)
       return unit;
    if (pin == t) {
      cb(0);
      count = 0; }
    else  {
      cb(-1);
      count = count+1; }
  }
}
```

```
object Attacker {
 Int attempt = 0;
 Int success = 0;

 Unit notify(Int r) {
   if (r == -1) {
     attempt = attempt+1;
     Acc.test(attempt,notify);
     Acc.test(success,notify);
  }
  else {
    success = attempt;
  }
 }
}
```

Secure compilation: conclusions

- Compilation techniques that preserve contextual equivalence address Key Challenge 1
  - The programming language is supposed to isolate the programmer from details of the platform to which the code is compiled
    - It is now OK to reason about security in terms of the source code
- We discussed how to do this for compiling towards the x86 platform
- The same idea is being explored for other platforms
  - Including so-called “multi-tier” languages for the web platform
  - This requires substantial additional machinery

Overview

- Introduction
- Secure compilation to native code
- Secure browsers
  - The browser is the new OS
  - What security architecture should it offer?
- Conclusions

Introduction

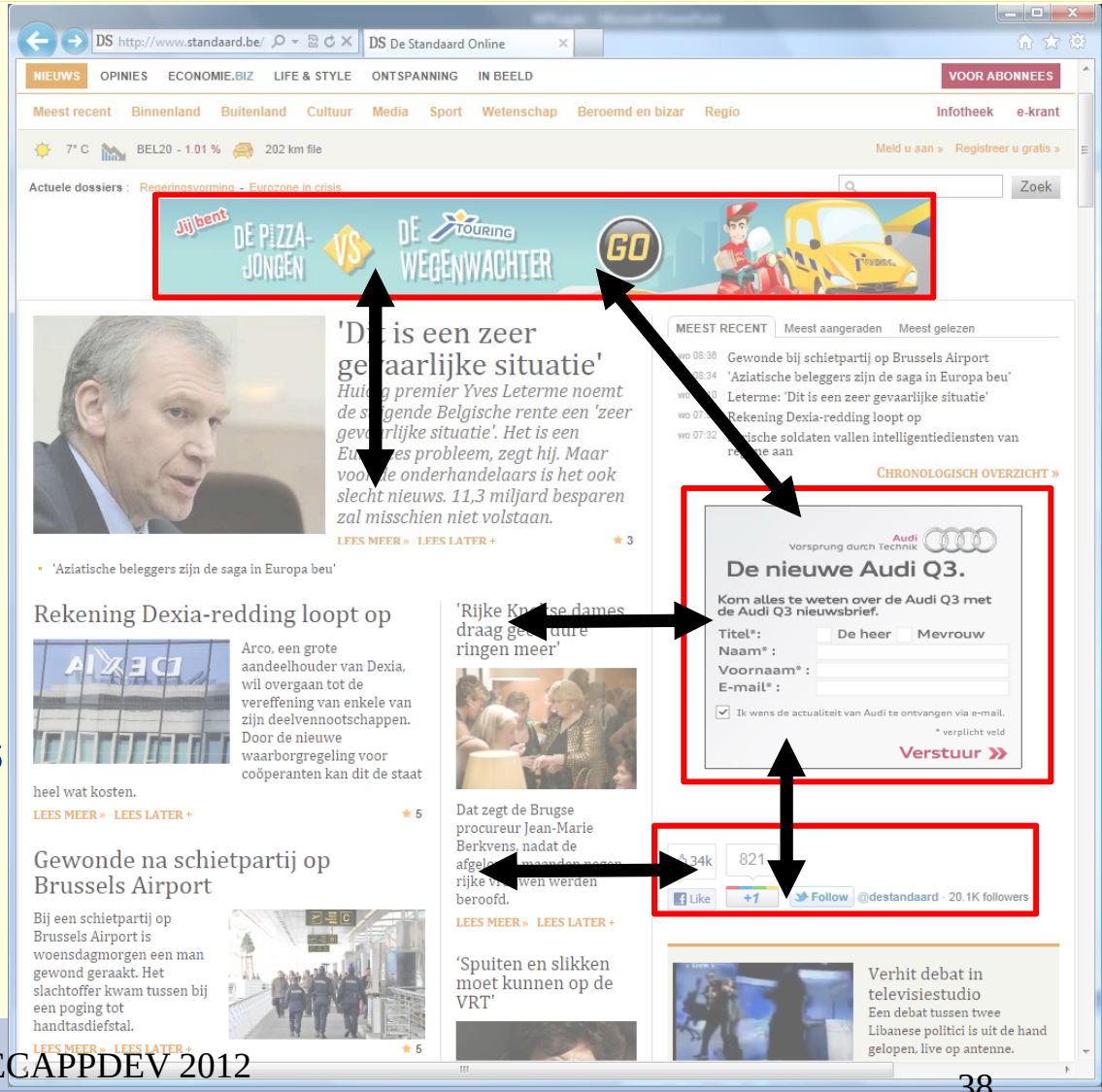
- Let's look at Key Challenge 2:
  - The platform is supposed to provide basic security guarantees to applications
- Modern operating systems were built to isolate multiple users
  - But most PC's (and definitely mobile devices) are single user
  - One single process on that OS is by far the most exposed and most security-critical component
    - And it has (almost) no benefit from OS-provided isolation

Introduction

- The browser handles content (data and executable code) from a variety of stakeholders
  - Multiple open tabs
  - Mashups within a single tab
- The browser implements isolation by means of the Same Origin Policy
  - Origin = (protocol, domain, port)
  - Ad-hoc restrictions are imposed on interactions between content from different origins

Third-party JavaScript is everywhere

- Advertisements
  - Adhese ad network
- Social web
  - Facebook Connect
  - Google+
  - Twitter
  - Feedsburner
- Tracking
  - Scorecardresearch
- Web Analytics
  - Yahoo! Web Analytics
  - Google Analytics
- …

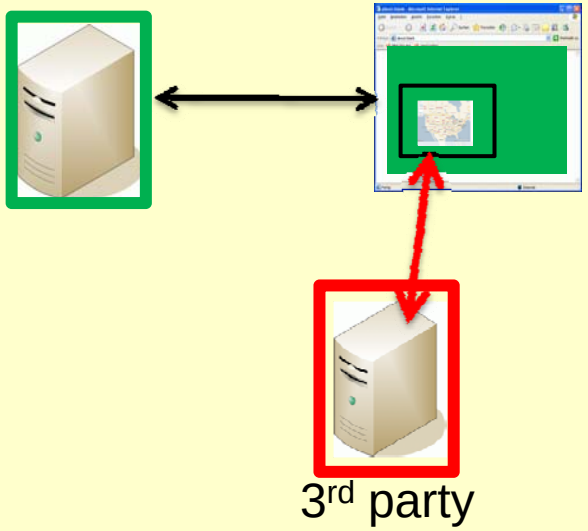


Integration of third-party JavaScript

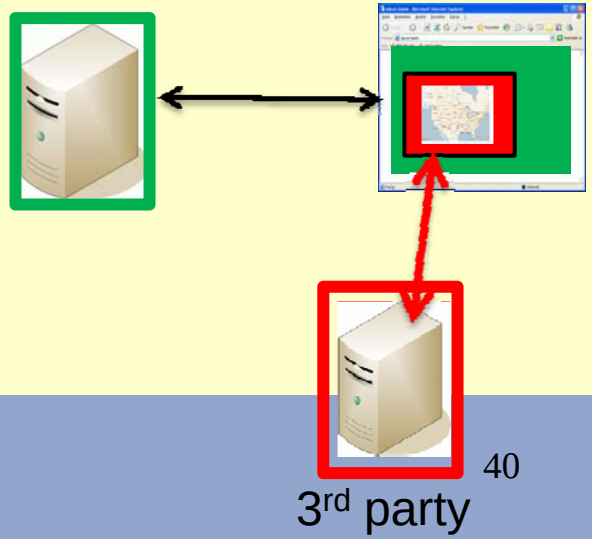
- Two basic composition techniques
  - Script inclusion
    - Third-party script run's in the execution context (i.e. origin) of the embedding page
    - Script has access to all the sensitive operations in this context
  - (Sandboxed) iframe integration
    - Third-party component runs in a separate security context (i.e. the origin of the third-party service provider)
    - Isolation between service provider and embedding page is realised via the Same-Origin Policy

Script inclusion vs iframe integration

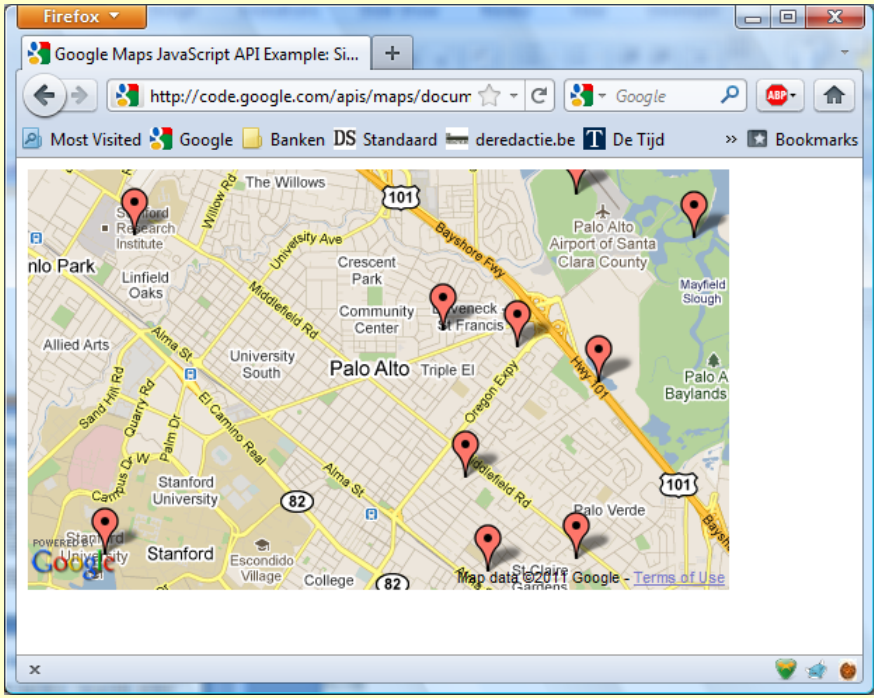
```
<html>
<body>
…
<script src="http://3rdparty.com/script.js">
</script>
…
</body>
</html>
```




```
<html>
<body>
…
<iframe src="http://3rdparty.com/frame.html">
</iframe>
…
</body>
</html>
```



Example: Google Maps integration



- Scenario:
  - User enters name of a location
  - GPS lookup via Google Geocoding API
  - Marker placed on the map via Gmap API

Google Maps code example

Script inclusion

Glue code

DOM element (div)

Summary

- A browser renders a complex mix of data and code from many stakeholders
- The Same-Origin-Policy and existing isolation techniques for scripts tend to favor insecure mixing of scripts
- In addition, script-injection vulnerabilities (XSS) may allow attackers to inject malicious scripts in the mix

Security and privacy consequences

- A large-scale empirical study presented at CCS 2010 shows that this is a **real** problem
  - Several popular sites (including Alexa global-top 100 sites) use JavaScript to violate user privacy by:
    - Stealing cookies
    - History sniffing
    - Behavior tracking
  - Note that these attacks are *invisible* to the user

Dongseok Jang, Ranjit Jhala, Sorin Lerner, Hovav Shacham,
An empirical study of privacy-violating information flows in JavaScript web applications, CCS 2010

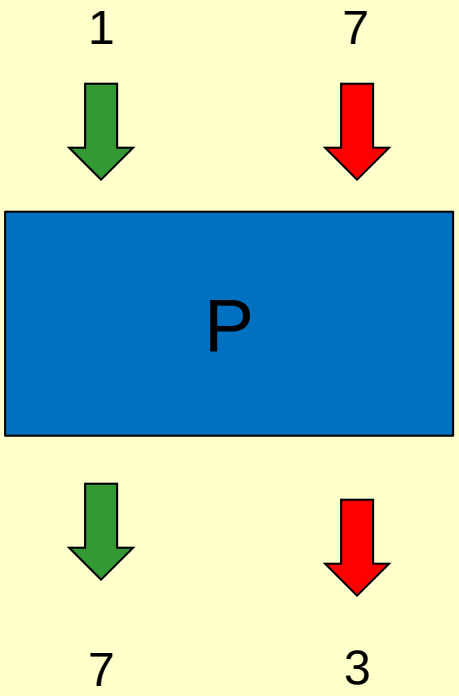
A better browser security architecture

- So what kind of security architecture is required from the browser?
  - It should protect user data confidentiality and integrity
  - In the presence of (possibly malicious) code handling that data
  - And it should be “compatible” with the current web

Information flow control to the rescue?

- Information flow control studies the enforcement of policies such as:
  - "Secret data should not leak to public channels"
  - "Low integrity data should not influence high-integrity data"
- A base-line policy (usually too strict – needs further relaxing) is non-interference:
  - Classify the inputs and outputs of a program into high-security and low-security
  - The low-outputs should not "depend on" the high inputs
  - More precisely: there should not exist two executions with the same low inputs but different high outputs

Illustration: non-interference



Secure:
Out_low := In_low + 6

Insecure:
Out_low := In_high

Insecure:
```
if (In_high > 10) {
    Out_low := 3;
}
else Out_low := 7
```

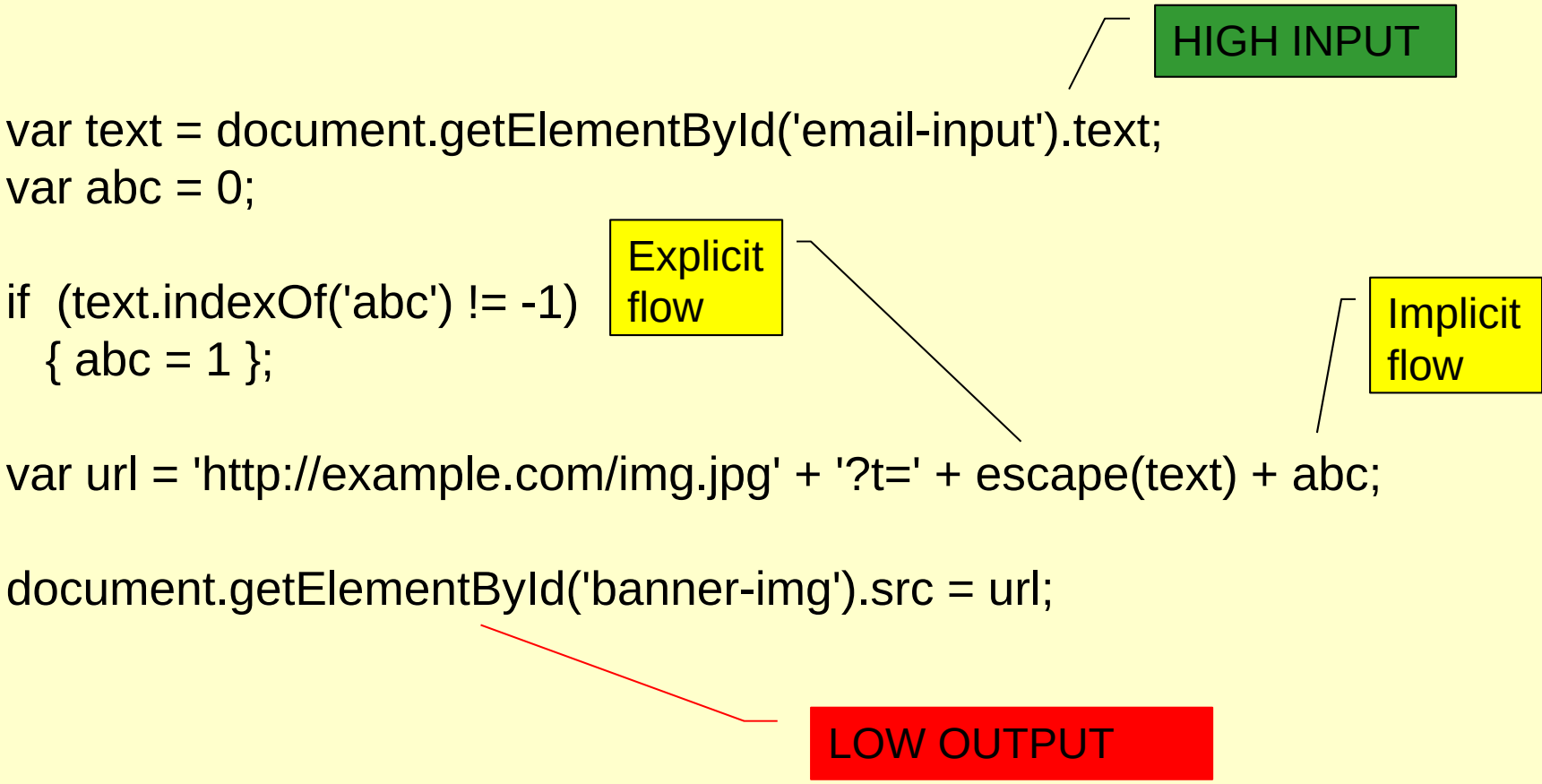
Example: information flow control in Javascript

HIGH INPUT

```
var text = document.getElementById('email-input').text;
var abc = 0;

if  (text.indexOf('abc') != -1)
   { abc = 1 };

var url = 'http://example.com/img.jpg' + '?t=' + escape(text) + abc;

document.getElementById('banner-img').src = url;
```

LOW OUTPUT

Example: information flow control in Javascript



Enforcing non-interference

- Static, compile-time techniques
  - Classify (=type) variables as either high or low
  - Forbid:
    - Assignments from high expressions to low variables
    - Assignments to low variables in “high contexts”
    - ...
- Two mature languages:
  - Jif: a Java variant
  - FlowCaml: an ML variant
- Experience: quite restrictive, labour intensive
  - Probably only useful in high-security settings

Enforcing non-interference

- Runtime techniques
  - Label all data entering the program with an appropriate security level
  - Propagate these levels throughout the computation
  - Block output of high-labeled data to a low output channel
- Several mature and practical systems, but all with remaining holes
- Some sound systems, but too expensive

Enforcing non-interference

- Alternative runtime technique: secure multi-execution
  - Run the program twice: a high and a low copy
  - Replace high inputs by default values for the low copy
  - Suppress high outputs in the low copy and low outputs in the high copy
- First fully sound and fully precise mechanism
- But obviously expensive
  - Worst-case double the execution time or double the memory usage

Dominique Devriese, Frank Piessens,
Noninterference through Secure Multi-execution,
IEEE Symposium on Security and Privacy, 2010

```
1 var text = document.getElementById
2         ('email-input').text undefined;
3 var abc = 0;
4 if(text.indexOf('abc')!=-1) { abc = 1 };
5 var url = 'http://example.com/img.jpg'
6     + '?t=' + escape(text) + abc;
7 document.getElementById('banner-img')
8     .src = url;
```

(a) Execution at L security level.

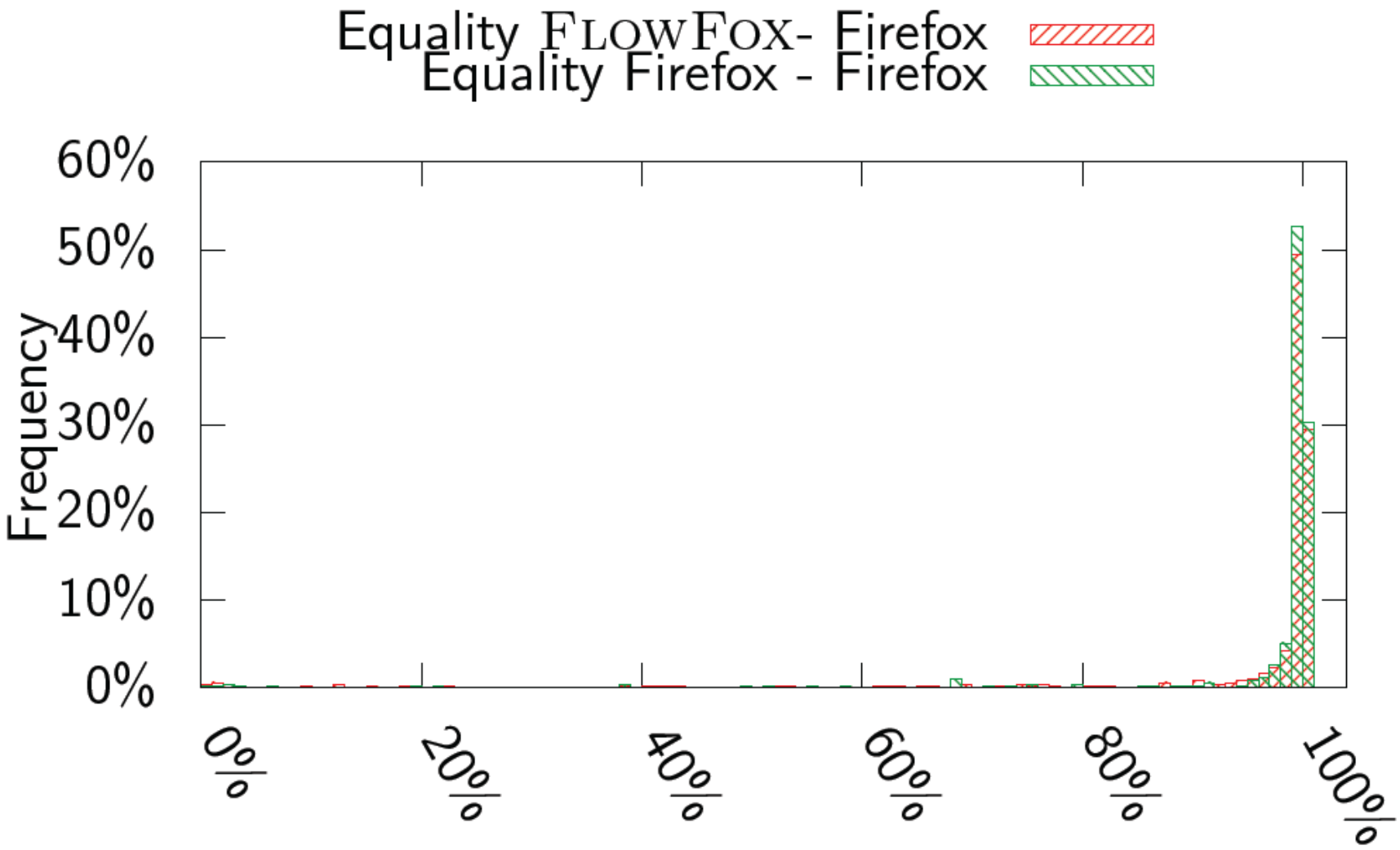
```
1 var text = document.getElementById
2         ('email-input').text;
3 var abc = 0;
4 if(text.indexOf('abc')!=-1) { abc = 1 };
5 var url = 'http://example.com/img.jpg'
6     + '?t=' + escape(text) + abc;
7 document.getElementById('banner-img')
8     .src = url;
```

(b) Execution at H security level.

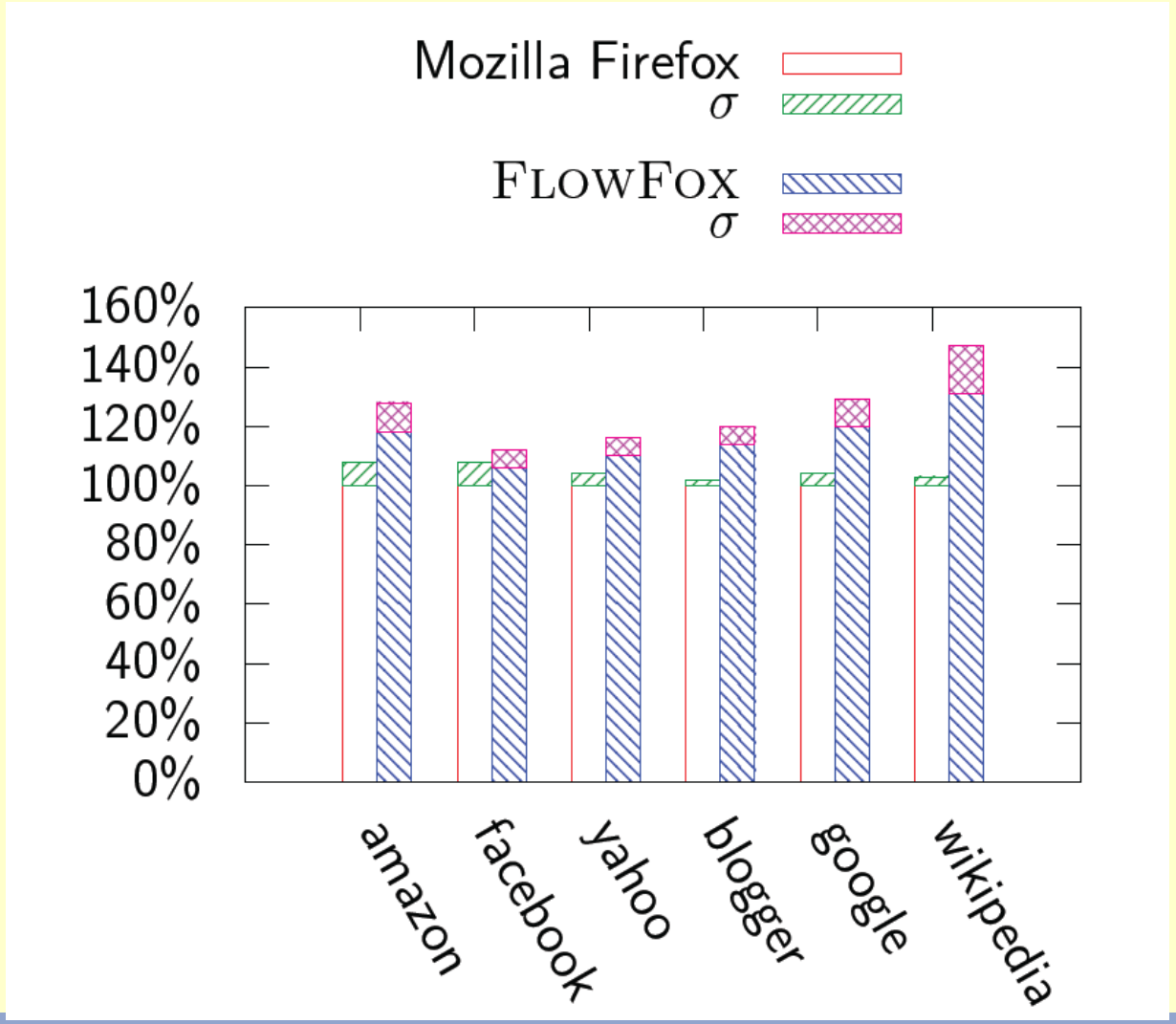
Does it work in a real browser?

- FlowFox is a variant of Firefox that implements information flow control for scripts by secure multi-execution
  - Implemented en evaluated by Willem De Groef as part of his PhD thesis
- Evaluation:
  - Is it “compatible” with the web?
  - Is it efficient?

Compatibility



Performance macro benchmarks



Secure browsers: Conclusions

- The current isolation mechanism implemented in browsers (the “same-origin-policy”) has important flaws
- Yet, this isolation mechanism is one of the key security mechanisms offered by the web platform.
- Understanding the security guarantees that should be offered by browsers is an important challenge for the coming years:
  - The browser as a “service-OS”
  - How securely share/divide real-estate on the screen?
  - Privacy protection
- Information flow control could be an important ingredient of the solution

Overview

- Introduction
- Secure compilation to native code
- Secure browsers
- Conclusions

Conclusions

- We have come a long way in improving software security
  - Process improvements
  - Coding guidelines
  - Tooling
  - …
- But rethinking platform security can substantially simplify things
  - Can we get rid of low-level vulnerabilities?
  - Can the platform provide generic, useful security guarantees?