
Controlling the execution of untrusted programs using high-level policies

Andrew J. Brown
A.J.Brown@cs.bham.ac.uk

School of Computer Science
University of Birmingham

CSRG talk :: November 13, 2006



- 1 Background and motivation
- 2 Policy enforcement via program monitoring
- 3 High-level policies to control programs
- 4 Related work, future work, and conclusions



Introduction

Users can trust few, if any, of the programs that they download.
Purposefully malicious programs can be classified by the term ‘Malware’.

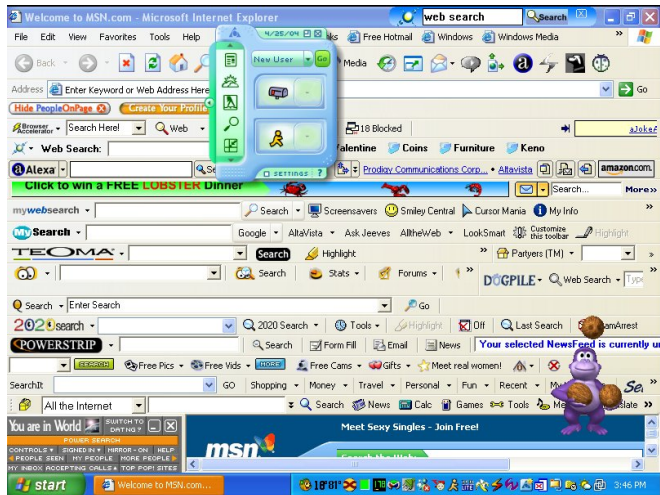
What is Malware?

- **Malware** is software intended to intercept and take partial control of a computer’s operation, without the user’s full consent. It subverts the operation of the host computer in order to benefit a third party.
- **Spyware** is another term often used to describe malware. It suggests that a program surreptitiously monitors the user. However, it has come to refer more broadly to any kind of malware.

Programmer-introduced vulnerabilities can make a program potentially malicious, without the programmer having malicious intent.



Malware infestation



The purpose of malware

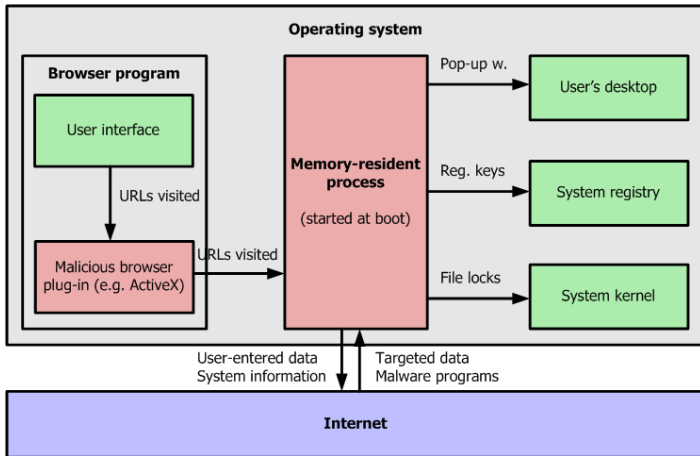
Attackers use malware to partially control a user's computer:

- To subject the user to targeted advertising
- To monitor the activity of the user
- To collect and hide illicit material
- To launch a DDoS attack on another service
- To commit fraud (e.g. keylogger stealing credit card details)
- To spread spam
- To spread *fear, uncertainty and doubt*.

Any program downloaded by the user is potentially unsafe and should not be trusted by the user.



Malware operation



Current malware prevention technologies

Attestation mechanisms: help user to trust an executable

- **Digitally-signed code**
Sign executable hash with private key and verify with public key
- **Hash functions**
Verify whether an executable has been altered

Harnessing mechanisms: do not require user to trust executable

- **Proof-carrying code**
Code carries a 'proof' with it to verify its integrity
- **Firewalls**
Prevents network connections other than those explicitly permitted
- **Anti-virus software**
Pattern matching on executables against definitions



Shortcomings of malware prevention technologies

Attestation mechanisms: help user to trust an executable

- **Digitally-signed code**
User must trust the author of the program
- **Hash functions**
The hash value may also be compromised

Harnessing mechanisms: do not require user to trust executable

- **Proof-carrying code**
Malicious program authors will not supply proofs
- **Firewalls**
Network traffic can be disguised as HTTP packets
- **Anti-virus software**
Inaccurate virus definitions; *recovery* simply deletes executables



Motivations

- 1 To allow end-users to safely run potentially malicious programs
 - User may *need* to run a program, despite potential risks
- 2 To allow users to remove malicious behaviour from programs during execution
 - Without impairing program performance greatly
 - Without suitably fine-grained policies
 - Without ability for program to subvert policies
- 3 To allow non-programmers to define / manage security policies
 - Reasoning about program behaviour without programming knowledge
- 4 To provide languages and tools that are high-level, yet offer a suitable level of granularity for policy specification



Foundations

Many properties can be used to reason about security (*privacy, secrecy, fairness, authentication, ...*). When reasoning about untrusted programs, we generally consider two properties:

Safety property

A predicate \hat{P} includes a safety property on a program S , whose action set is \mathcal{A} , iff: $\forall \sigma \in \mathcal{A}^*. \neg \hat{P}(\sigma) \Rightarrow \forall \sigma' \in \mathcal{A}^*. \neg \hat{P}(\sigma; \sigma')$

Liveness property

A predicate \hat{P} includes a liveness property on a program S , whose action set is \mathcal{A} , iff: $\forall \sigma \in \mathcal{A}^* : \exists \sigma' \succeq \sigma : \hat{P}(\sigma')$

... where σ is a single finite execution



Enforcing policies to prevent malware

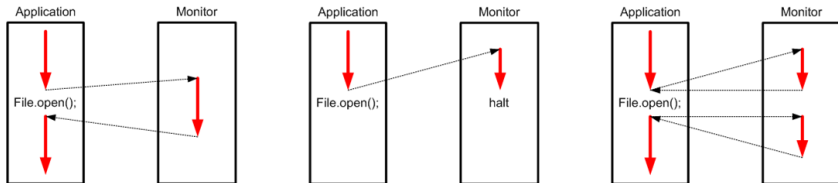
Language-based security mechanisms protect a host from untrusted applications by analysing or modifying program behaviour. Techniques for implementing language-based security fall into two distinct categories:

- **Static mechanisms:** Analysis at link-time, include; *type checking, proof checking, model checking*
- **Dynamic mechanisms:** Analysis at runtime (cf. runtime verification), include; *runtime monitoring, reference monitoring, stack inspection*
- What information about a program can we monitor at runtime?
 - **System calls:** Monitor / censor system calls a program makes (e.g. sandboxing)
 - **Control stack:** Monitor program's current state; analyse control stack (e.g. JVM)
 - **Library calls:** Monitor every program action (e.g. AOP)



Runtime monitors

A runtime monitor is a program that runs in parallel with a target program and examines actions invoked by that target's instruction stream.



Runtime monitors ensure that only safe code is executed:

- Detect, prevent and recover from program errors at runtime
- Make decisions which may be based on execution history
- Have no knowledge of future application actions



Polymer: Creating runtime monitors from policies

A language and system for composing and implementing security policies on untrusted Java programs [Ligatti, Walker, and Bauer 2004]. Polymer allows:

- First-class policies, specified separately from target source code:
 - Actions are Java method calls with arguments
 $\alpha \equiv \text{package.Class.method} < \text{init} > (\text{args})$
 - Actions are monitored prior to their execution by the Java virtual machine
 - $\alpha \in A$ (A is finite / countably infinite)
- Higher-order policy combinators:
 - *conjunction, selection, precedence, modification*
- Policies to be execution transformers:
 - If actions violate a policy, Polymer can transform bytecode
 $\{\alpha_w; \alpha_x; \alpha_x; \alpha_y\} \text{ monitor } \{\alpha_w; \alpha_x; \alpha_y; \alpha_z\}$



Polymer: example policy

Program under consideration may not read files from or write files to any storage device after it has connected to a network port

```
privateActionSet actions =
    new ActionSet(new ActionPattern[]{
        < * java.io.FileInputStream.open.<init>(..) >,
        < * java.io.FileOutputStream.open.<init>(..) >,
        < * java.net.Socket.connect.<init>(..) > } );
public Suggestion query (Action a) {
    private boolean networkAccess = false;
    private RemoveWrite r = new RemoveWrite(this);

    System.out.println("Program being transformed: " + a);

    aswitch(a) {
        if (networkAccess == true) {
            case < * java.io.FileInputStream.open.<init>(..) >:
                return new ReplacementSuggestion(r, a);
            case < * java.io.FileOutputStream.open.<init>(..) >:
                return new ReplacementSuggestion(r, a);}
        else if (networkAccess == false) {
            case < * java.net.Socket.connect.<init>(..) >:
                networkAccess = true;
                return new OKSuggestion(this, a);}
    } }
```



Polymer: policy format

Polymer has a unique policy format that is *imperative*, and allows reasoning about program events. Its higher-order policy combinators allow one to compose **superpolicies** from **subpolicies**. Polymer achieves this with:

- *public class MyPolicy extends Policy*
 - Action specification
 - *package.Class.method<init>(args)*
 - Decision procedure
 - *public abstract Suggestion query(Action a)*
 - *aswitch(a){case <* package.Class.method<init>(args)>:}*
 - Enforcement procedure
 - *public void accept(Suggestion s){if(s.isOKSug()){...}}*
 - *public void result(Suggestion s, Object r, boolean e){}*
- Suggestions may be:
IrrSug, OKSug, InsSug, ReplSug, ExnSug, HaltSug



Polymer summary

Polymer has many useful features that aid policy specification. It allows:

- The safe execution of untrusted programs with some guarantees
- Recovery from policy violations at runtime
- First-class policies to be expressed
- A policy hierarchy to be created using higher-order combinators
- Reasoning about program events in an imperative language

Polymer cannot be used without programming knowledge:

- Policies for real-world programs approx. 1500 - 2000 lines
- Unclear how to express some policies
- Constructs that allow policy composition increase complexity



Our work

A 'user-oriented' sandbox with a simple declarative language for policy specification:

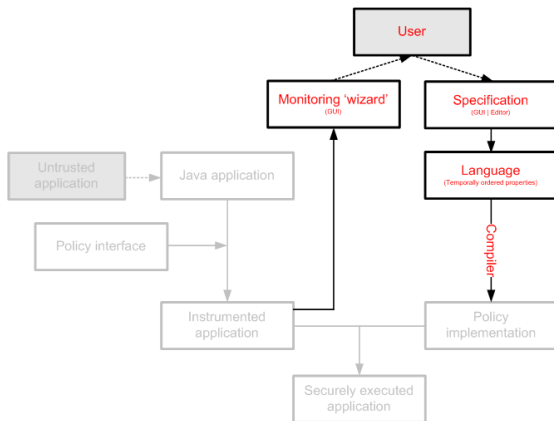
- Allow users to specify/compile security policy into Polymer (GUI-driven)
- Provide language constructs and compiler methods for:
 - Relating abstract actions to action sets
 - Expressing temporal order in policies
 - Selecting appropriate recovery actions in policies

Ease of specification and greater expressivity:

- High-level declarative constructs for basic policies
- Lower-level human-readable rules (cf. SQL)
- Ability to express properties that other (weaker) languages cannot
- A structured language to express transformation requirements



Our additions to Polymer architecture



Polymer architecture (with additions of this work)



Example: Browser policy

Policy 1:

PUC: May access a URL only if that URL was entered in a URL field by the user, or if that URL was hyperlinked from the page previously loaded.

SUPRESS $network.connect(urlString) \leftarrow \neg(gui.TextField.get(urlString) \vee gui.Page.contains(urlString))$

Policy 2:

PUC: May write to a directory only if that directory is the cookies directory, or it is the cache directory.

SUPRESS $file.write(path) \leftarrow \neg(filesystem.getDirProperties(cookiePath) \vee filesystem.getDirProperties(cachePath))$



Example: Browser attack

When a secure connection is made by a browser, a padlock is displayed. This informs the user that traffic between the browser and the server is encrypted (using HTTPS).

- What if a form is *secure*, but it posts back data *insecurely*?
 - The web designer does not understand the process of encrypting sessions (securing blank form; passing confidential data in the clear)
- How does a browser behave?
 - **Internet Explorer:** Warns the user about any switch between HTTP and HTTPS. Everyone turns this off!
 - **Mozilla Firefox:** Warns the user of this situation, regardless of whether they turned off standard warnings.
- What can we do?



Example: Browser policy

Policy 3:

PUC: Must always send data using HTTPS between initiating a secure connection and the user being warned that they are leaving that secure connection.

EXCEPTION $\leftarrow (\neg \text{network.Connect}(\text{"HTTP"})) \text{ AFTER}$
network.Connect("HTTPS") UNTIL gui.Dialog.connectionWarning()



Specifying properties in a high-level language

When running inside 'harnessed' execution environment:

Untrusted program(A),

For actions (α) (Entity.operation(attributes(s))) that:

Match All: \wedge

Match Any: \vee

Of the following conditions:

α_x	OCCURS NEVER OCCURS ATMOST n TIMES OCCURS ATLEAST n TIMES OCCURS ALWAYS (OCCURS = < >)	FROM NOW BEFORE α_y AFTER α_y BETWEEN α_y AND α_z AFTER α_y UNTIL α_z
------------	--	---

Select a Polymer Suggestion to instantiate, from the following:

{IrrSug, OKSug, InsSug, ReplSug, ExnSug, HaltSug}

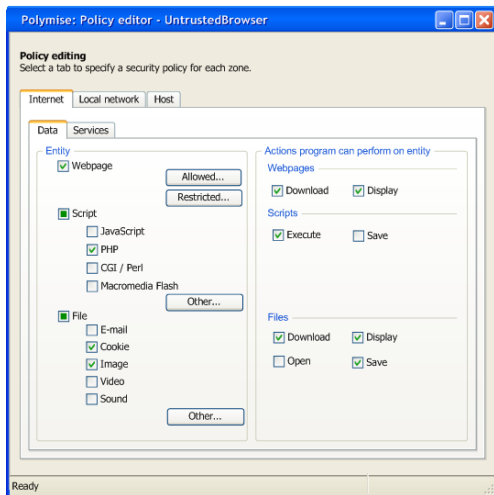


Higher-level policy specification

- This policy specification level is still considered too low-level for most end-users:
 - A fine-grained declarative language mediation of programs
 - LTL foundations allow specification of temporal order
 - Sentence construction is *almost* intuitive
- A higher-level construct for simpler policy specification:
 - PUC can interact with zones {internet, network, host}
 - Zones can have resources {data, services, devices}
 - PUC can perform abstract actions on resources



Higher-level policy specification



Problems

- Our work uses patterns [Dwyer et al. 2000] for finite-state property specification
- Patterns build upon LTL, which is too weak to specify some policies
- We cannot write context-free policies ($\forall x \exists y$) using LTL:
 - *“Every file read or file write must be preceded by a file open, and followed by a file close”*
- Neither can we write context-sensitive policies using LTL:
 - *Object : Object has type PasswordTextField*



Future work

Problems:

- Policy cannot specify whether data was entered in a particular frame
- A password may have been entered legitimately in some other frame
- A user must enter action sequences using Java syntax
 - Abstract Action compilation requires a clear system ontology
- Are context-free / context-sensitive constructs actually required?

Initial solutions

- Use information flow principles to track variable assignments throughout execution
- Create a complete ontology for the core Java API that can be reconfigured by the user
- Monitor between bytecode and JVM, map concepts to lower-level commands



Conclusion

- Current attestation and harnessing mechanisms do not allow sufficient mediation
- End-user mechanisms for policy enforcement are not sufficiently granular
- What if the end user wants to run the program? Finer-grained control required
- Polymer allows this: difficult to write security policies
- Languages for policy specification are not suitably expressive
- Our work hopes to build an expressive, user-operable system



Questions and comments

Thank you for listening! Your comments would be much appreciated

