

# Secure Programming Lecture 16

## Software Protection

David Aspinall

Informatics @ Edinburgh

# Outline

## Overview

Man-At-The-End Attacks

Defences

- Code signing

- Obfuscation

- Tamperproofing

- Watermarking

Summary

# Recap

We have looked at:

- ▶ examples of vulnerabilities and exploits
- ▶ particular programming failure patterns
- ▶ security software engineering
- ▶ tools: static analysis for code review
- ▶ language-based security

In this lecture we look at the special case of adding *security features to protect software* itself.

# Outline

Overview

**Man-At-The-End Attacks**

Defences

Code signing

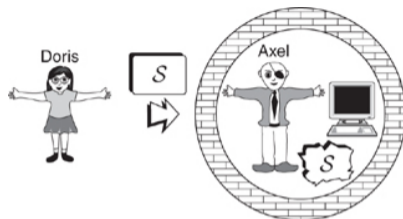
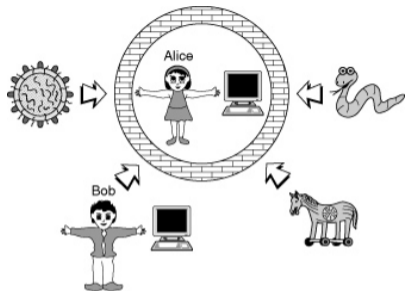
Obfuscation

Tamperproofing

Watermarking

Summary

# Inside vs outside threats



- ▶ Bob wants to attack Alice
- ▶ Security perimeter stops bad things getting in or information leaking out

- ▶ Axel wants to attack Doris
- ▶ Security features must protect how digital assets are used

Note: *insider threats* more generally refers to threats on left picture when insiders deliberately violate security policy.

# MATE and R-MATE

## Man-At-The-End (MATE) Attacks

An adversary has physical access to a device and compromises it by inspecting, reverse engineering or tampering with its hardware or software

## Remote Man-At-The-End (R-MATE) Attacks

In distributed systems where untrusted clients communicate with trusted servers, a malicious user gets an advantage by compromising an untrusted device.

It would be better to use gender neutral names like **Person-at-the-end** and **Remote Person-at-the-end** just as **Middle Person** is replacing “man-in-the-middle”.

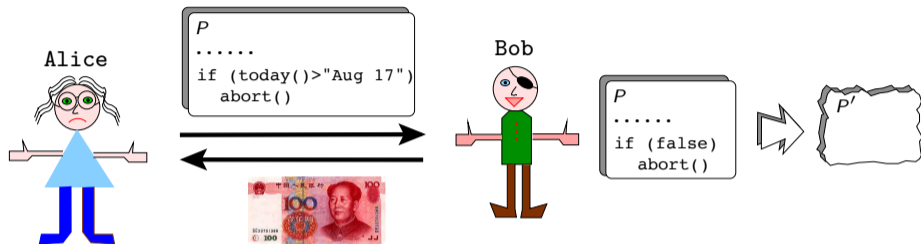
# MATE attack scenarios

Example goals of an attacker include:

1. Software piracy
2. License check removal
3. Malicious reverse engineering
4. DRM key extraction
5. Protocol discovery
6. Violation of export/supply chain controls

Mostly, these attack *non-availability* where there is an attempt to deny access to an asset, e.g., unless it has been paid for.

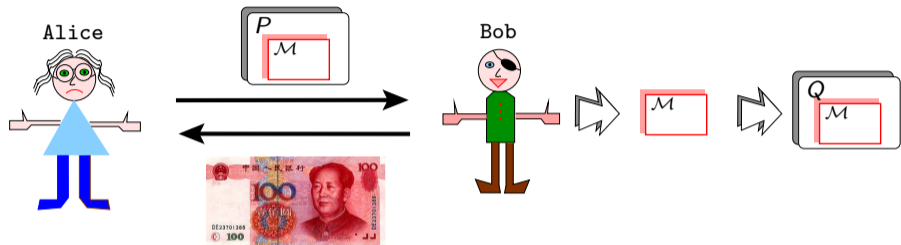
# License check removal



- ▶ Alice sells a time-limited license for her software
- ▶ Bob removes the license check to use it indefinitely
- ▶ Defence: Alice makes her program *tamperproof*



# Malicious reverse engineering



- ▶ Alice's  $P$  has a trade secret algorithm  $\mathcal{M}$
- ▶ Bob copies  $\mathcal{M}$  into his program ("code lifting").
- ▶ Defence: Alice *obfuscates* her code to make reverse engineering difficult

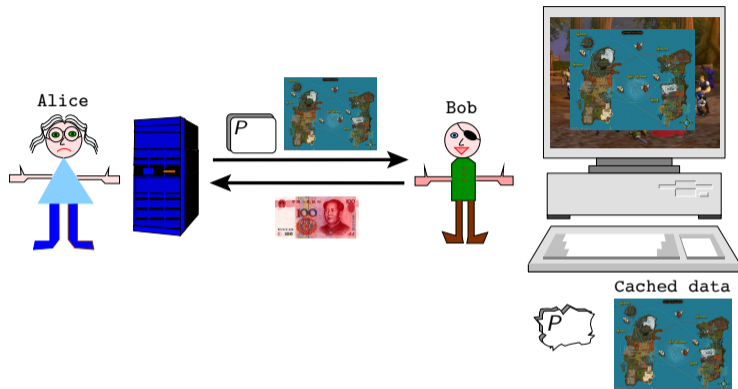
# R-MATE attack scenarios

Example goals of an attacker include:

1. Cheating in networked computer games
2. Accessing or altering distributed medical records
3. Attacking wireless sensor networks
4. Hacking smart meters to disrupt supply

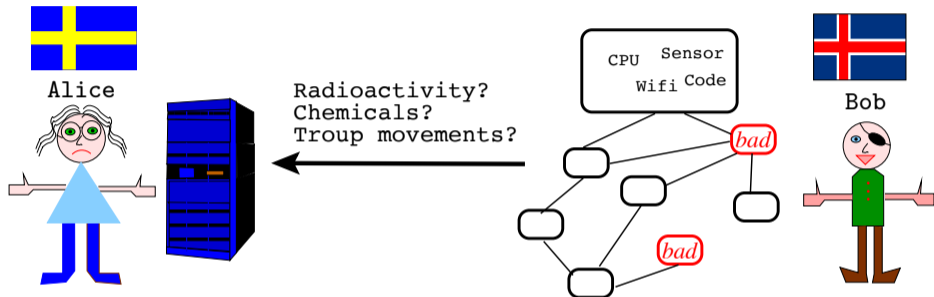
These examples attack confidentiality and integrity as well as availability.

# Cheating in networked computer games



- ▶ Alice runs online game with paid-for inventory
- ▶ Bob re-uses cached data. Advantage: free resources
- ▶ Defence: *replay-resilient protocol*

# Wireless sensor network attacks



- ▶ Alice collects data from a wireless sensor network
- ▶ Bob interferes with some of the sensors
- ▶ Defences: *anomaly detection, remote attestation.*

# Outline

Overview

Man-At-The-End Attacks

**Defences**

Code signing

Obfuscation

Tamperproofing

Watermarking

Summary

# Outline

Overview

Man-At-The-End Attacks

**Defences**

**Code signing**

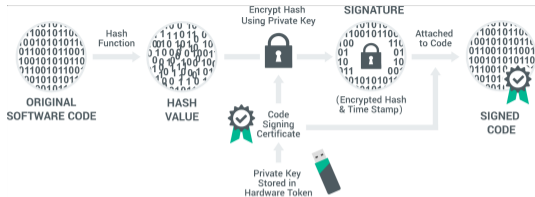
Obfuscation

Tamperproofing

Watermarking

Summary

# Code signing

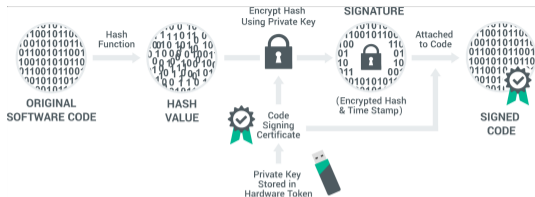


- ▶ Cornerstone for code integrity and authenticity
- ▶ Detects tampering before code execution
- ▶ Aims to protect recipient from unsafe code (especially malware)

With a trusted (secure) platform we could also protect against MATE.

**Question.** How would that work?

# Code signing



- ▶ Cornerstone for code integrity and authenticity
- ▶ Detects tampering before code execution
- ▶ Aims to protect recipient from unsafe code (especially malware)

With a trusted (secure) platform we could also protect against MATE.

**Question.** How would that work?

**Question.** What two vulnerabilities can defeat code signing?



# Outline

Overview

Man-At-The-End Attacks

**Defences**

Code signing

**Obfuscation**

Tamperproofing

Watermarking

Summary

# Program obfuscation

Here is a program burton1.c:

```
char o,o[];main(l){for(;~l;0||puts(o))0=(0[o]=  
~(l=getchar())?4<(4^l>>5)?l:46:0)?--0&printf("%02x ",l)*5:!0;}
```

What does this program do?

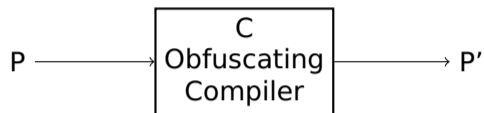
# Program obfuscation

```
bash-3.2$ gcc burton1.c --no-warnings -o burton1
bash-3.2$ echo "David" | ./burton1
44 61 76 69 64 0a David.
```

This is a winner from the 2018 [International Obfuscated C Code Contest](#). It won as the [best one-liner](#), a judge noted: *One line, one array, one loop and one statement but it prints many bytes. It won.*

Similar contests are available in other languages. Most entries are intended as fun puzzles, rather than serious efforts at software protection.

# Obfuscating compilation



An obfuscating compiler  $C$  transforms a program  $P$  into a *functionally equivalent* program  $P'$ .

The idea is that  $P'$  conceals the code of  $P$  to be “inscrutable” so an attacker cannot learn information about its operation (algorithms, or embedded data such as cryptographic keys).

**Question.** What properties should  $C$  have?

## Practical obfuscation

Many intuitively obfuscating techniques can be used:

- ▶ Rename identifiers. Use equivalent expressions (e.g., bit shift multiply)
- ▶ Code and data: reorder, duplicate, add dummies
- ▶ Flatten control flow (e.g., use jump tables)
- ▶ Merge and split functions (inline, outline)
- ▶ Introduce pointer aliases. Add concurrency.
- ▶ Use *opaque predicates*.
- ▶ Use a custom abstract machine

Combining these transformations can make human understanding hard and thwart automated code analysis. Various obfuscation tools (commercial, non-commercial) are available.

**Exercise.** Which operations are *guaranteed* to make analysis difficult?  
(hard question)

# Obfuscation in theory

## Black box simulator

The *black-box simulator*  $S^P$  of  $P$  can only observe the input-output behaviour of  $P$ , nothing about its code or timing.

$C$  is a good obfuscator if an attacker cannot learn anything more about  $P$  by examining  $C(P)$  than from its black-box behaviour  $S^P$ .

## Virtual Black-Box Security

$C$  is *virtual black box secure* if for all  $P$ , for all attacks  $A$  which examine the obfuscation  $P' = C(P)$ , then  $A(P') \simeq_c A(S^P)$ .

$\simeq_c$  is *computational indistinguishability* which restricts to computationally bounded attackers whose power depends on a security parameter (e.g., limiting runtime).

# The power of general obfuscation

In fact, many cryptographic primitives can be *derived* from obfuscation.

## Symmetric to Asymmetric Crypto

Given a secret key  $K$  and symmetric encryption function  $E_K$ , publish its obfuscated version  $C(E_K)$ . Thus anyone can encrypt but only the owner who knows  $K$  can decrypt.

## Homomorphic Encryption

Homomorphic encryption allows general computation on encrypted data. For any boolean operation  $f$ , the plain program  $P$  computes  $E_K(f(D_K(x)))$ . Its obfuscated version hides the key  $K$  and encryption method.

## Impossibility of (general) obfuscation

Given the power of a perfect obfuscation primitive, it isn't surprising that it is hard to find one. . .

A celebrated result of Barak et al shows that it is *impossible* to construct an obfuscating compiler that satisfies virtual black box (VBB) security. The proof uses a counterexample, based on a program *Secret* which recognises a hidden secret value  $S$  and also recognises its own code, outputting  $S$ ; its VBB obfuscation must hide the secret but also necessarily reveal it!

This result doesn't mean that we can't have an obfuscating compiler that works on *some* programs. Also, VBB is a very strong requirement, weaker notions like *Indistinguishability Obfuscation* may be enough in practice.

Barak et al. *On the (Im)possibility of Obfuscating Programs*. CRYPTO 2001. See Chapter 5 of [Surreptitious Software](#) for a listing of a program *Secret*.



# Outline

Overview

Man-At-The-End Attacks

**Defences**

Code signing

Obfuscation

**Tamperproofing**

Watermarking

Summary

# Software Tamperproofing

Tamperproofing aims to make sure a program executes as intended by its author, even when the person running may try to disrupt, monitor or change execution.

Two aspects, captured as functions in code (or environment):

- ▶ **Check** to see if tampering has occurred
- ▶ **Respond** somehow, imposing a penalty

The penalty might be to exit the program or degrade its operation.

**Question.** How is this requirement stronger than integrity checking with a code hash?

# Tamperproof Checking

Checking may examine different things:

- ▶ **Code checking:** has program been changed?
- ▶ **Result checking:** is result of computation correct?
- ▶ **Environment checking:** are we running in a debugger?

# Tamperproof Responding

Responding may take different actions:

- ▶ **Termination**
- ▶ **Restore** recover the program to intended state
- ▶ **Degrade** return incorrect results; slow down operation
- ▶ **Report** phone home to owner
- ▶ **Punish** destroy program, data or environment

In each the first 4 cases, response may be designed to be stealthy to avoid alerting user (or helping attacker defeat system).

## Pervasive Hashing

One technique is to use multiple hashing methods and compute multiple hashes on fragments of code.

Then spread the hash computation repeatedly throughout the code.

To help prevent attackers figuring out the scheme, tamperproofing is combined with obfuscation.

The Skype VoIP clients used this technique.

# Outline

Overview

Man-At-The-End Attacks

**Defences**

Code signing

Obfuscation

Tamperproofing

**Watermarking**

Summary

# Digital Watermarking



Digital Watermarking hides one digital signal inside another, perhaps covertly (i.e., invisibly) exploiting properties of the human visual system.

The “Lena” image was a standard test image used in early days of digital image processing research, this is from an influential 1996 paper (reference at end). It is not an acceptable image to use nowadays!

# Software Watermarking



A watermark embedding function transforms a program  $P$  into a watermarked program  $P_w = embed_k(P, w)$ .

- ▶ A secret key  $k$  is needed to guide embedding
- ▶ The watermark should be recoverable, perhaps probabilistically, to someone who has the key  $k$ , or more widely if intended.
- ▶ It should be *robust* (not removable)
- ▶ Have *high credibility* (low FP, FN accuracy)



# Applications

Various applications of watermarking, depending on what data is embedded:

- ▶ **Track Authorship:** copyright owner
- ▶ **Track Purchaser:** purchaser/licensee (fingerprinting)
- ▶ **Record Rights:** usage restrictions
- ▶ **Integrity:** cryptographic hash of code

The last case is essentially the same as code signing.

# Watermarking techniques

Numerous methods:

- ▶ Embed meta data directly (strings in code)
  - ▶ use “opaque predicates” to make robust
  - ▶ or code signing
- ▶ Encode information in obfuscation operations
  - ▶ e.g., permutation of code blocks
- ▶ Use a public **blockchain** to record metadata, hash values
  - ▶ Non-Fungible Tokens (NFTs)

## Using opaque predicates

```
public class Fibonacci {  
    public int fibonacci (int n) {  
        String copyright = "Copyright (C) by Clever Coders, Inc";  
        if complexTest()  
            n = length(copyright);  
        if (n <= 2)  
            return 1;  
        else  
            return fib(n-1) + fib(n-2);  
    }  
}
```

An opaque predicate is one whose value is a constant, known to the programmer, but is not obvious from the code and so must be computed at run time.

# Outline

Overview

Man-At-The-End Attacks

Defences

- Code signing

- Obfuscation

- Tamperproofing

- Watermarking

**Summary**

# Summary

Protecting software and information usage rights is often required and a range of countermeasures have been developed.

- ▶ **Code signing:** cryptographic assurance about integrity and origin
- ▶ **Obfuscation:** raises attacker effort for reversing
- ▶ **Watermarking:** visibly/invisibly trace software/data
- ▶ **Tamperproofing:** detect modification and abort
- ▶ **Hardware root of trust:** lock code/execution to a device (TPMs etc)
- ▶ **Advanced crypto:** compute on encrypted data

The last two methods are both becoming more practical and in future may replace (or augment) earlier ones.

# Review Questions

## **Software protection defence methods**

- ▶ Compare and contrast each of the defence measures.
- ▶ Do any of the methods improve or detract from the situation with potentially exploitable code vulnerabilities?

## **Software protection attack methods**

- ▶ What methods are open to the attacker against each of the defences listed?
- ▶ Discuss the potential use of cryptographically oriented attacks against software protection methods.

## References and credits

Most of this lecture is based on the 2009 Addison-Wesley book:

- ▶ [Surreptitious Software](#) by Christian Collberg and Jasvir Nagra.

Pictures of the attack scenarios are used with permission from Christian Collberg's slides.

The image of code signing is from Comodo's page [How Does EV Code Signing Work?](#).

The example of watermarked images is from the paper *Transparent Robust Image Watermarking* by Swanson, Zhu and Tewfik, International Conference on Image Processing, Sept 1996.

## Further reading

Collberg's book *Surreptitious Software* is currently out-of-print and the University's subscription to InformIT comes and goes. But you can read some of Christian Collberg's papers here:

- ▶ [https://scholar.google.co.uk/citations?user=lq\\_d12gAAAAJ&hl=en&oi=ao](https://scholar.google.co.uk/citations?user=lq_d12gAAAAJ&hl=en&oi=ao)

Here is a 2017 literature review from on some of the techniques mentioned:

- ▶ <https://doi.org/10.1016/j.infsof.2018.07.007>

(Mainly useful for pointers, but the overview of tool aims and methods in Section 4 and Tables 2 and 3, Fig 11 are interesting to look at.)