# Security Analysis of BOINC

**Karl Chen, Paul Huang**

Department of Computer Science – University of California, Berkeley

`email: {quarl, pbhuang}@cs.berkeley.edu`

## Abstract

Berkeley Open Infrastructure for Network Computing (BOINC) is a software platform for distributed-computing using volunteered computer resources. It generalizes the software-engineering aspects of SETI@home-type projects; it's the ”@home” in SETI@home. We analyze why the security defenses in BOINC are of paramount importance, discuss possible attacks on BOINC, and propose and analyze defenses. We also compare some other distributed-computing products.

## Motivation

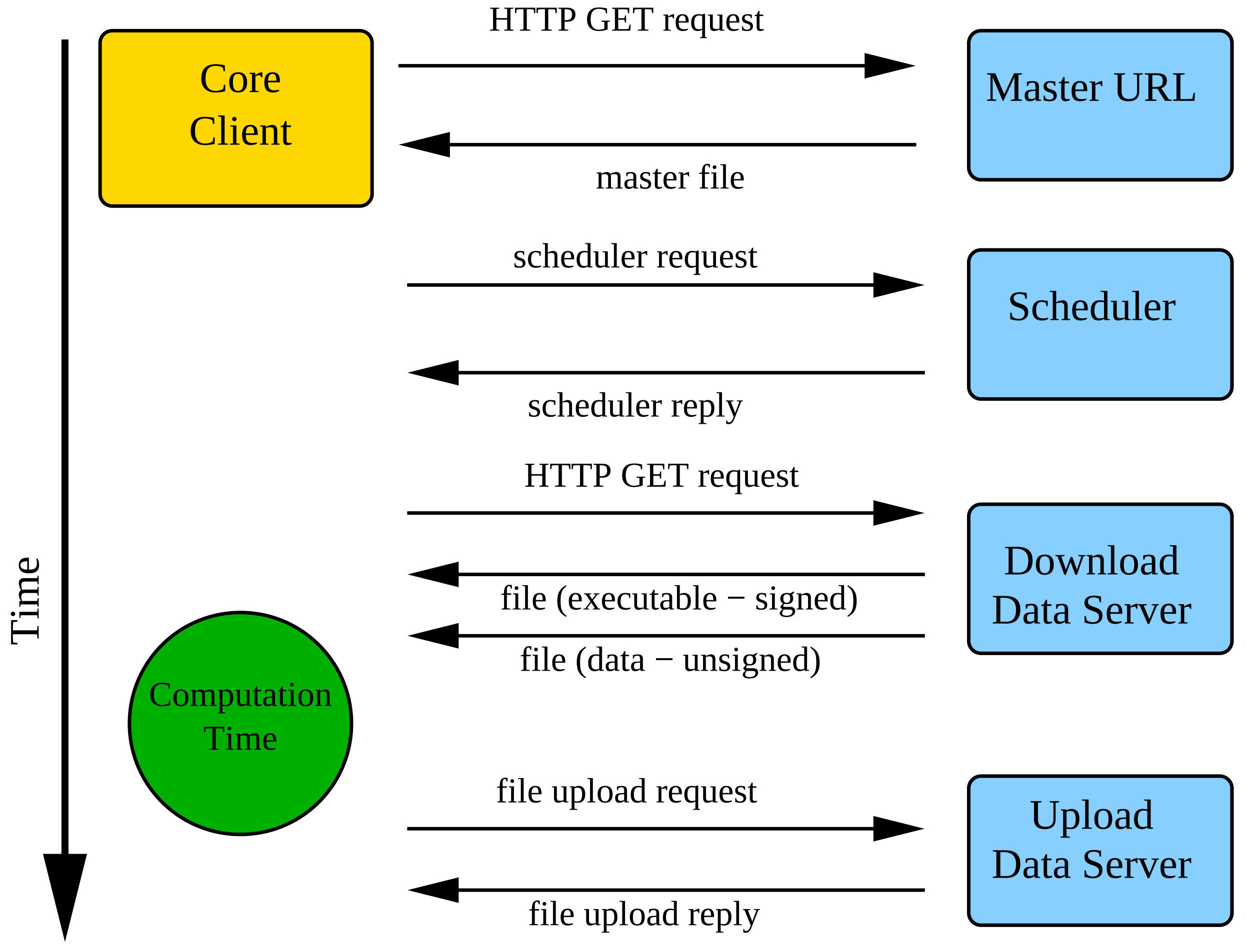
Why is compromise of a BOINC server more dangerous than compromise of www.mozilla.org?

- 4 million users running the client software (soon)
- Automatic, periodic download and execution of binary applications; default interval of 1 day
- Compromised server ≈ malicious servers
- BOINC clients assume non-malicious servers
- Compromise unnoticed for 1 day ⇒ all clients compromised
- Mass-compromises of clients (stealing credit card numbers, deleting files) can lead to the end of public distributed computing

### Worms possible

- Attack on server + attack on client ⇒ worm!
  - Clients are 99% Windows/x86 monoculture
  - Servers are Linux/Solaris running Apache + MySQL + PHP + BOINC
- Hop over firewalls, infect non-public servers
- Servers also know clients' IP addresses & OS versions

## BOINC Architecture Overview



Core Client

HTTP GET request

master file

Master URL

scheduler request

scheduler reply

Scheduler

HTTP GET request

file (executable − signed)

file (data − unsigned)

Download Data Server

Computation Time

file upload request

file upload reply

Upload Data Server

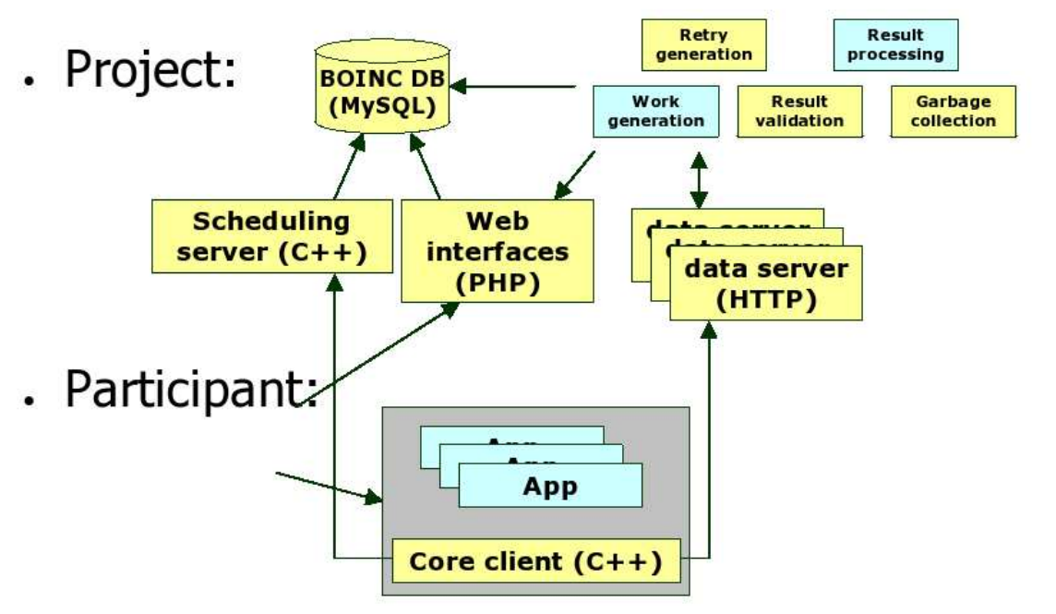
Time

### Server components

- Master URL
- Schedulers
- Download Servers
- Upload Servers

Non-user-visible:

- Database, daemons



. Project:

BOINC DB (MySQL)

Retry generation

Result processing

Work generation

Result validation

Garbage collection

Scheduling server (C++)

Web interfaces (PHP)

data server (HTTP)

. Participant:

App

Core client (C++)

## Threat Model

- 1 million computers could be compromised in 1 day
- Attackers may invest high resources
  - Man-in-the-middle attacks
  - Find holes in any services running on BOINC servers

## Current Defenses

- Binary executables are signed using RSA
- Server components can be separated
- Uploading files requires signature (to prevent DoS on upload server)

## Insufficient!

General problem of running untrusted applications; signatures not the best solution. If private key compromised:

- Keys don't have lifetime (violates fail-safe defaults)
- Re-keying protocol: sign new key with old key
- Only creates a race between project managers and attacker to re-key - and attacker has a head start

Not only executables can be exploited (violates principle of complete mediation):

- Applications use non-type-safe languages (so far)
- Applications invoke external programs
- Complexity ⇒ security verification difficult
- Applications in the past have had security holes
- Attack on input files ⇒ arbitrary code execution
  - "`tar`": attacker could pass "`--rsh-command`" if not careful
  - "`tar x`", embedded zip library: no check for "`../`" when extracting
    - ∗ Overwrite arbitrary files, including executables

### Attacks on input files:

- Client follows pointers from Master URL → Schedulers → Download servers
- Compromise anywhere in path ⇒ attack on input files
  - Vulnerable to DNS attacks, MITM attacks
  - Master URL contains embedded XML list of schedulers, as well as "user of the day" profile, news, etc. (violates the principle of least common mechanism)

## Analysis of Other Architectures

### Some other architectures:

- MoneyBee
- Classic Folding@Home
- UnitedDevices
- Distributed.net
- D2OL

### Other architectures:

- Most lack feature of downloading executables (less insecure)
- Most have less levels of indirection to download (less insecure)
- Most download unsigned input files (equally insecure)
- Some rely on secret key embedded in application (insecure)
- Many rely on security through obscurity (insecure)
- One uses Java (very secure)

### GRID architectures:

- Servers and clients generally assume each other trusted, so little security

## Defenses

### Specific issues:

- `tar`, Zip library: add checks for paths
- Master URL: don't re-use "bells-and-whistles" page
- Compromise of input files: sign them also
- Projects can choose to use type-safe language

### Treat application as untrusted:

- Sandbox each application: feasible, partial solution (chroot)
- Interpreted language: not feasible in short term (legacy applications)
- Virtual machine: not feasible (performance)
- Proof-carrying code: feasible, but need it for external programs as well (one-time compiler cost)
- Intercept and mediate syscalls: feasible (Janus for Solaris, Linux)