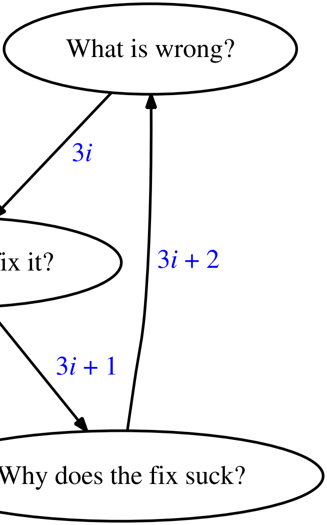

What is wrong?
3i
How can we fix it?
3i + 1
Why does the fix suck?
3i + 2

kcgi: sandboxed CGI framework

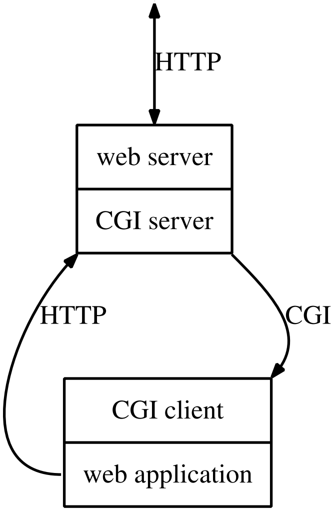
Or: quantifying the price of web application security

AsiaBSDCon 2015, Tokyo

Kristaps Dzonsons
`kristaps@bsd.lv`, `kristaps@kcons.eu`
BSD.lv Project, k-Consulting

May 28, 2015

Part I: i_0: web application security


HTTP
web server
CGI server
HTTP
CGI
CGI client
web application

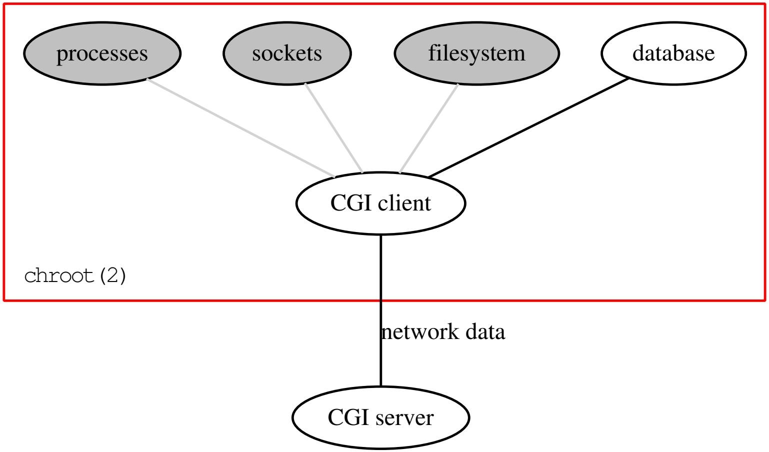
1. Client passes HTTP request to web server.
2. Web server receives HTTP request from client.
3. . . . maps request to CGI.
4. . . . spawns CGI script.
5. . . . passes HTTP request to CGI script.
6. CGI script processes.
7. . . . passes HTTP response to server.
8. Web server passes HTTP response to client.

6. CGI script processes.

. . . is really. . .

6.
    6.1 CGI script parses header request information.
    6.2 . . . reads request into memory/file.
    6.3 . . . parses key-value pairs from request stream.
    6.4 . . . processes key-value pairs.

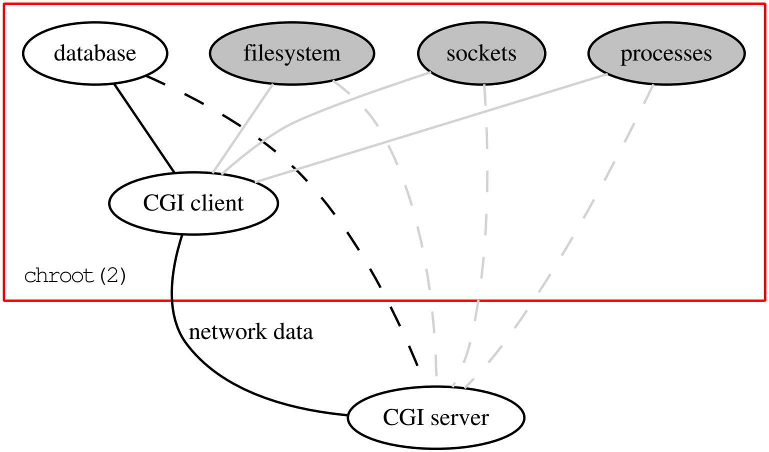
1. Client passes HTTP request to web server.
2. Web server receives HTTP request from client.
3. . . . maps request to CGI.
4. . . . spawns CGI script.
5. . . . drop privileges and `chroot(2)` child.
6. . . . re-write components of HTTP request into CGI.
7. . . . passes HTTP request to CGI script.
8. CGI script processes.
9. . . . passes HTTP response to server.
10. Web server post-processes HTTP response.
11. Web server passes HTTP response to client.


processes
sockets
filesystem
database
CGI client
chroot(2)
network data
CGI server

Problem i_0: adversarial network data directly in contact with system resources.

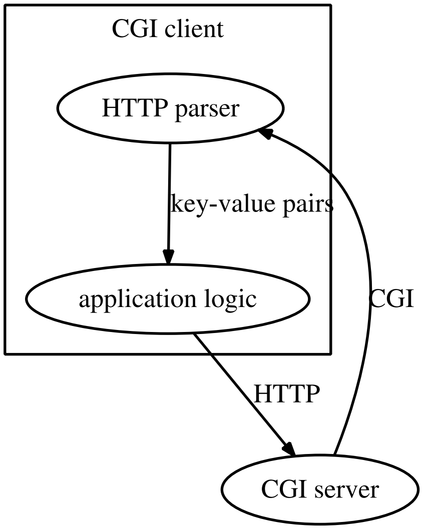
- Database and application-specific resources.
- System resources (sockets, processes, ...)
- File-system within `chroot(2)`.
- Memory of application (!).
- ...
- *All* your base.

CAPTAIN! WHAT YOU SAY !!


database
filesystem
sockets
processes
CGI client
chroot(2)
network data
CGI server

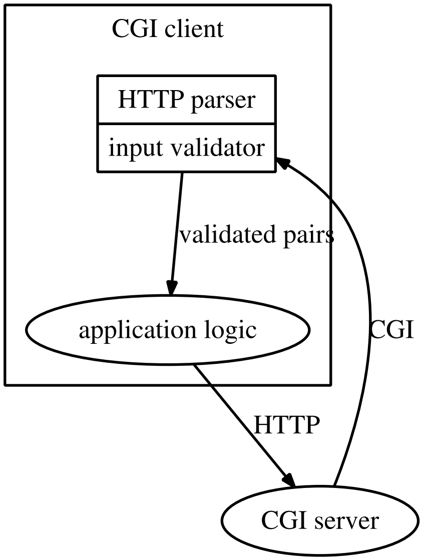
Specifically, problem i_0 is connection of application logic with the code that parses HTTP form data (and HTTP environment) from the CGI request.

Solution $i_0 + 1$?


CGI client
HTTP parser
key-value pairs
application logic
CGI
HTTP
CGI server

By splitting apart the parser, we can protect web application logic (the process) from requests exploiting the parser.

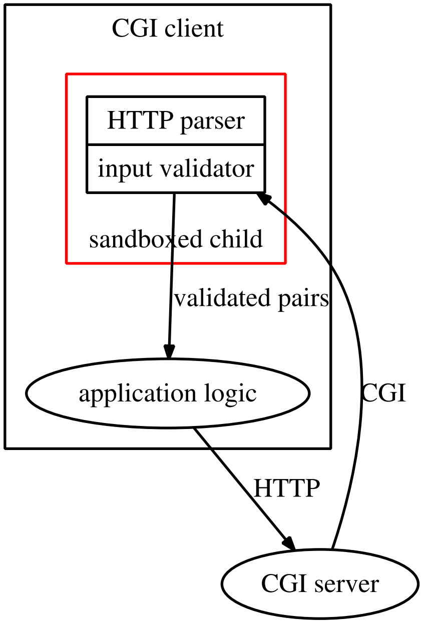
We can do better. . .


CGI client
HTTP parser
input validator
validated pairs
application logic
CGI
HTTP
CGI server

1. CGI script spawns child processes *before* anything else.
2. . . . hands off standard input to child.
3. . . . sets up socket pair with child.
4. Untrusted child parses header request information.
5. . . . reads request into memory/file.
6. . . . parses key-value pairs from request stream.
7. . . . passes key-value pairs back to CGI script.
8. CGI script processes key-value pairs.

The untrusted child can still access system resources.

We can do better. . .


CGI client
HTTP parser
input validator
sandboxed child
validated pairs
application logic
CGI
HTTP
CGI server

Sandboxing is a way of constraining the environment available to a process.

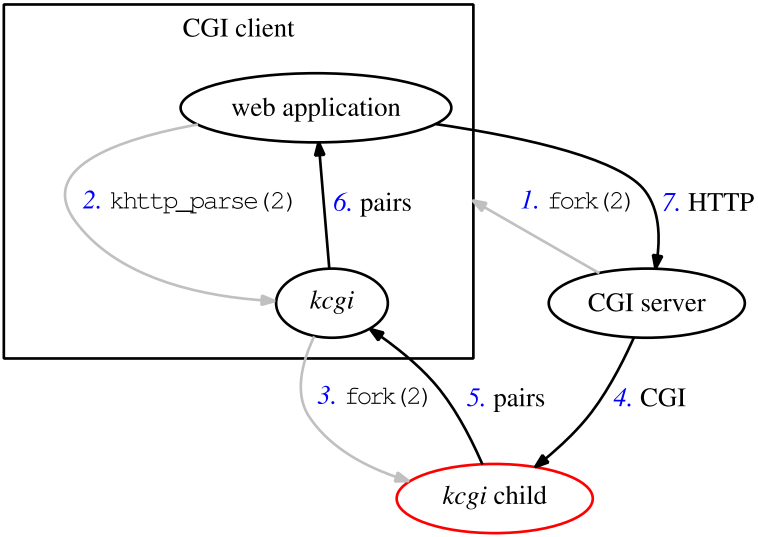
Most well-known sandbox? `chroot(2)`.

More thorough sandboxes: `systrace(4)` (OpenBSD), Capsicum (FreeBSD), “sandbox” (Darwin), `ed(1)`, . . .

By sandboxing the parse sequence, we limit the damage caused by untrusted network data. As for what the application logic does with that data…You're on your own.

Now on to $i_0 + 2$ and i_1…

Part II: $i_0 + 2$: *kcgi*


CGI client
web application
2. khttp_parse(2)
6. pairs
1. fork(2)
7. HTTP
kcgi
CGI server
3. fork(2)
5. pairs
4. CGI
kcgi child

kcgi, `kristaps.bsd.lv/kcgi`, is a C library that is[1]. . .

1. designed to fail
2. slow
3. resource-intensive

[1]It also has manpage documentation, a regression framework, AFL testing framework, automatic HTTP compression, and considerable MIME parsing.

kcgi, `kristaps.bsd.lv/kcgi`, is a C library that is. . .

1. designed to fail
   1.1 parse and validate *everything* in child process
   1.2 sandbox following OpenSSH's example
2. slow
   2.1 at least twice as slow as raw parsing
3. resource-intensive
   3.1 one extra process per CGI client
   3.2 two extra file descriptors
   3.3 reads full request into memory twice

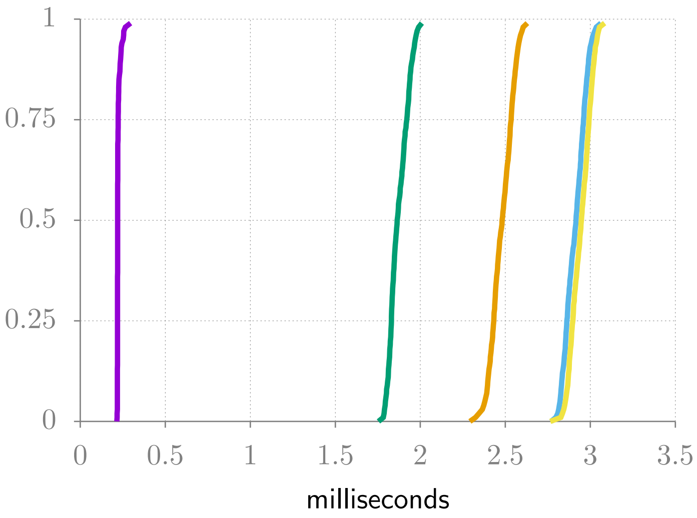


Figure : OpenBSD 5.5, nginx, `slowcgi(8)`

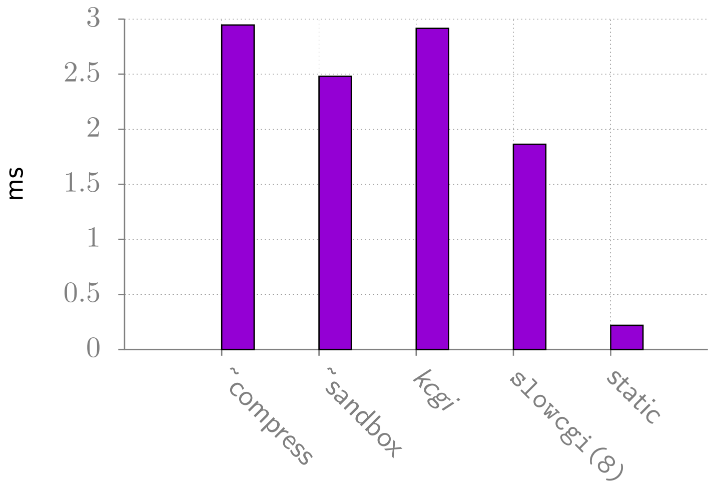


Figure : OpenBSD 5.5, nginx, `slowcgi(8)`

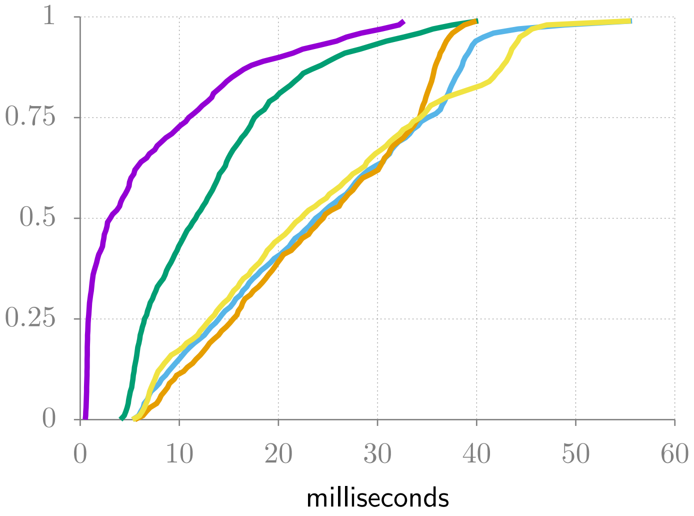


Figure : Mac OS X “Lion”, Apache

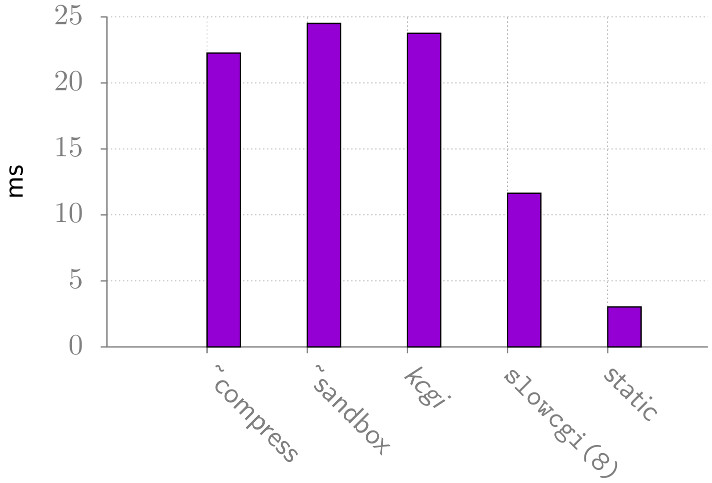


Figure : Mac OS X “Lion”, Apache

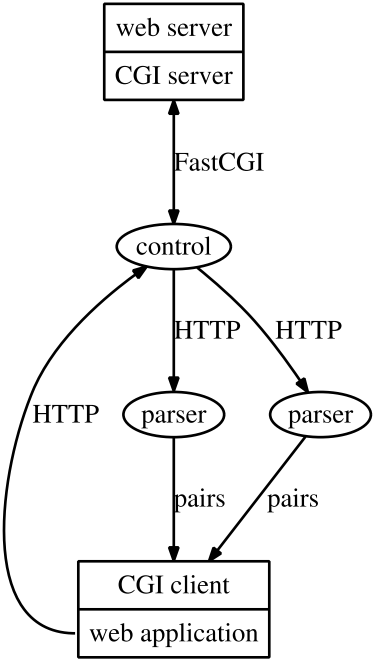
kcgi suffers greatly from the penalty of `fork(2)`, `systrace(4)`, and `socketpair(2)` I/O. It also suffers from double-allocation of data (original for parse, parsed pairs in parent).

Problem i_1? Performance and resource usage.

The allocation problem can be improved by smart programming. But the former?

Solution $i_1 + 1$: FastCGI. This fixes the amount of forking.

This is a *work in progress*^w^w^w^w*future work*.


web server
CGI server
FastCGI
control
HTTP
HTTP
HTTP
parser
parser
pairs
pairs
CGI client
web application

While this is being finished... *kcgi* is available at `kristaps.bsd.lv/kcgi`.

Questions?

Thank you!