kcgi: securing CGI web applications

Kristaps Dzonsons
BSD.lv Project

Abstract

With privilege separation, sandboxes, capabilities, and jails, the BSD systems have gained increasingly sophisticated tools to protect developers from themselves. These tools constrain the resources available to processes whether operating normally or exploited by an adversary. In this paper, I'll introduce a library, *kcgi*, that brings these constraints to bear on a special class of applications: web applications. Web applications ("CGI scripts") are broad enough to have only one common characteristic: a complex, non-interactive input channel. Connected either directly to the Internet or proxied through a web server, they often expose precious resources in response to their input. *kcgi* helps by parsing and proxying input to calling applications in a sandboxed child.

1 Introduction

There are few computing environments more hostile than those of web applications. Graphical utilities, for example, require a button-pushing operator. Command-line utilities inherit login credentials. Web applications—like all network-facing applications—have no such protection: they must consider arbitrary input from arbitrary locations.

The potential for damage is proportionate. While a compromised graphical or command-line system might allow a local adversary access to privileged system access, a compromised network application avails itself to the Internet.

It's little wonder that network applications are subject to intense scrutiny, often taking advantage of the strongest in system protection. Processes often run as unprivileged users, "dropping" superuser credentials after startup, and run in a *chroot*, which limits file-system access to a sanitised root. This limits the scope of damage in the event of compromise.

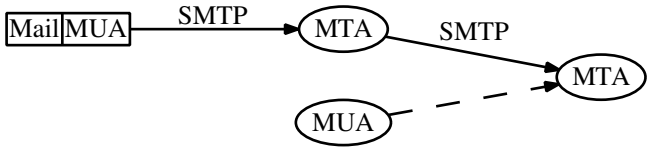
More advanced protection consists of breaking tools into separate purpose-specific components that communicate amongst themselves. This further limits the damage of exploits to only their component space and not the full chain of execution.

OpenSMTPD[1], for instance, is broken into nine separated processes, with each isolated to work only with data necessary to their task, isolated on the file-

[1] `https://www.opensmtpd.org`

system and running without special privileges. [2] OpenSSH[2] is also broken into separate processes for handling network data. [3, 5, 6]

These utilities focus on securing the *transport* logic, not the application logic. In the former, transport logic consists of the mail message envelope delivery, while application logic is the parsing and reading of the message contents. In the latter, transport consists of user input and output—not the utilities invoked on behalf of that user.



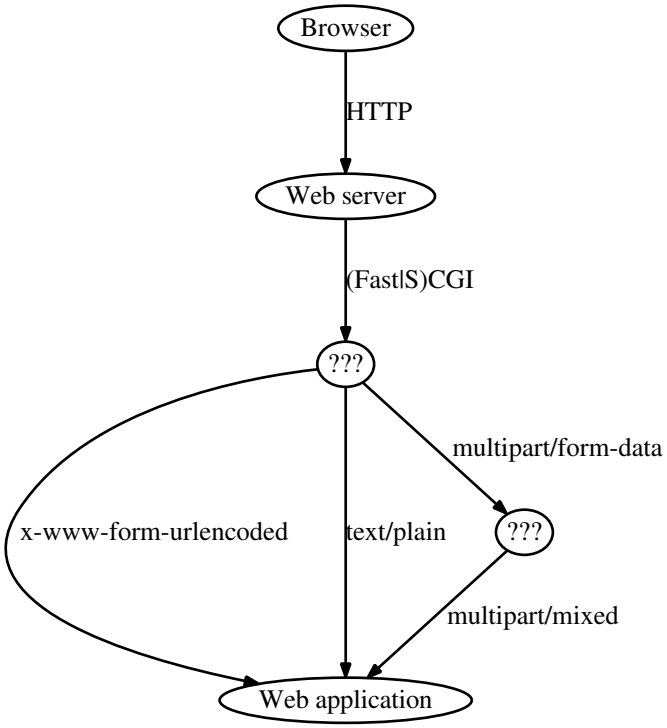
Can we do the same for web applications? It seems straightforward to apply these principles of separation. A web application uses HTTP as its transport protocol, usually encapsulated by CGI or its "fast" alternatives (FastCGI, SCGI, etc.). But the boundary between transport and application logic is difficult to define.

While OpenSMTPD can limit itself to SMTP proper, how much or how little of HTTP[3] can be considered "transport"? HTTP is simple enough, but any reasonable HTTP message consists of POST data of significant complexity.

For perspective, we consider the functionality of some popular tools used for parsing POST data for web applications: CGI(3p), cgi.py, and Tom Boutell's cgic. These parse the key-value pairs of generic HTTP POST data, which consists of the following formats.

[2] `http://www.openssh.org`

[3] For simplicity, we disregard the overhead of the CGI, FastCGI, etc. protocols.

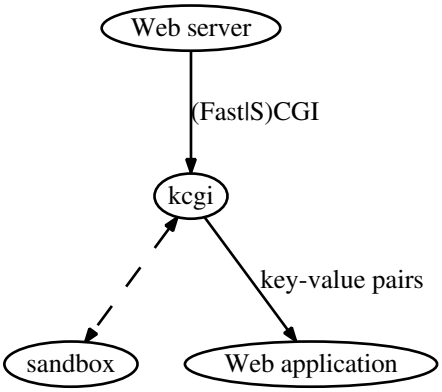


The multipart/form-data comprises the lion's share of complexity. The multipart/form-data standard, whose rules follow the MIME standard, inherits MIME's nesting. Since key-values may be defined in form-data documents nested within the outer MIME document, any POST processing must include them.

We then return to our tools to consider their security precautions, and additionally consider other C-based libraries: qDecoder, libcgic, cgi-c, etc. *None* of these tools use file-system or user privilege dropping, not even to mention component separation. All parsing occurs within the calling process's context. This means that untrusted network data is parsed within a trusted context. The job of privilege separation, if possible at all, is left as an exercise to the application developer.

In this paper, I introduce a system to address the shortcomings of handling untrusted network data in CGI clients by introducing privilege-separation to the "lowest common denominator" of web application transport: CGI/HTTP form parsing. This system, *kcgi*[4], is an ISO C library for handling the parsing and validation of HTTP request (form) data and environment for ephemeral and persistent CGI systems. It is ISC licensed. Unlike conventional frameworks, the library does so in a privilege-separated *and* sandboxed environment.

[4] `http://kristaps.bsd.lv/kcgi`



The goal of *kcgi* is to establish a contract between the web application and its input data. In short, when *kcgi* returns the key-value pairs of an HTTP request, the elements of that request have been validated according to the application writer's specification: a PNG file has been validated as a proper PNG, integers are proper integers, and so on. These validations occur in a privilege-separated and sandboxed process, so that malicious input intended to compromise validation, such as a bogus PNG file, will not compromise the web application context or other requests.

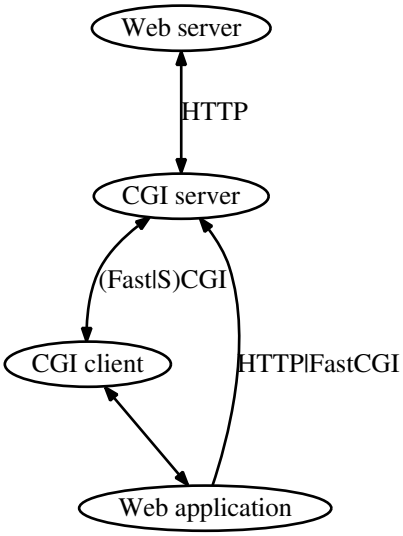
This departs from the common usage of CGI libraries, which serve only to extract POST data. *kcgi* can be used for this as well, of course.

The contract is enforced by a series of validators for each parsed key-value pair, where the parsing occurs in the sandboxed child and data is transferred over socket pairs. The sandbox is implemented in a system-dependent manner. For ephemeral CGI systems, this is fairly straightforward; for persistent CGI systems, it's quite complicated.

kcgi has been deployed for several ephemeral CGI applications since 2012. The persistent (FastCGI, SCGI, etc.) framework is still under development.

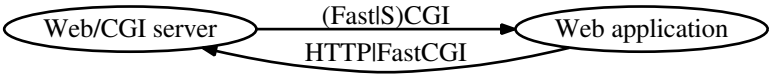
2 Model

Web applications consist of several logical components: the *web server*, which accepts requests; the *CGI server*, which accepts requests as configured by the web server; the *CGI client*, which accepts requests from the CGI server; and the *application*, which processes the request itself.



In terms of data, the web server minimally interprets the HTTP document to conditionally pass requests to the CGI server. The CGI server then re-formats into one of several possible formats: CGI, FastCGI, SCGI, etc. Data exchange between the CGI client and web application varies widely. Responses are passed from the web application to the CGI server or web server, either as HTTP to be re-purposed by the web server or in a CGI-level language like FastCGI, which is re-purposed by the CGI server.

Traditionally, the CGI server is incorporated into the web server process, while the CGI client is incorporated into the web application.



In *ephemeral* configurations, the CGI client and application are spawned per-request by the web server. In *persistent* settings, the CGI client and application are spawned independently of the web server and communicate over sockets.

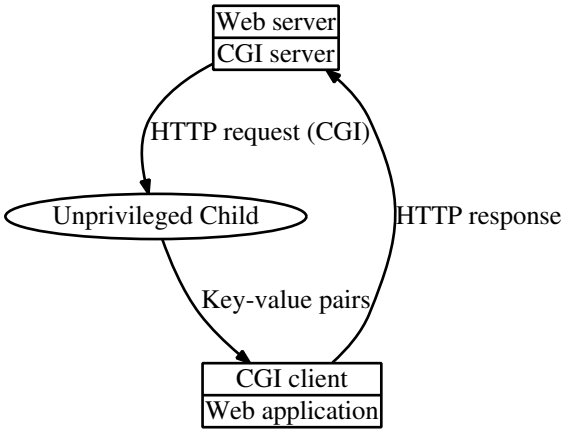
kcgi operates within or as the CGI client, accepting network data on behalf of or directly from the web server and providing it to the web application. Classically, the CGI client and web application are one physical entity; but in the event of *kcgi*, they are in fact separate processes.

2.1 Ephemeral Clients

An ephemeral CGI and web application instance is invoked by the web server for each HTTP request. Data from the server flows to the instance according to the CGI protocol, RFC 3875[5]. The client's responsibility is to parse the HTTP document into key-value pairs and environment and pass the pairs to the web application. The web application must then orchestrate an HTTP response,

[5] `http://www.ietf.org/rfc/rfc3875`

which it passes directly to the CGI component (parent process) of the web server.

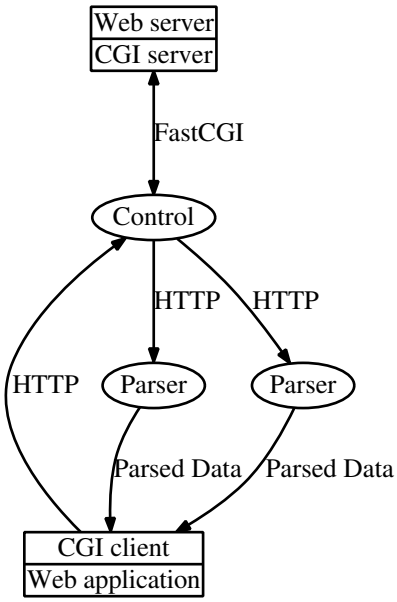


The *kcgi* CGI client spawns a child process when it is invoked via a function call from the web application, which should occur as early as possible to minimise inheritance "bleed" from the parent. This child process then reads and processes the CGI server's CGI-formatted HTTP request data (multipart/form-data, etc.), validates its pairs in the child process, then returns parsed and validated key-value pairs and environment to the parent over a trusted socket pair. The web application then orchestrates its HTTP response from the validated data.

2.2 Persistent Clients

Persistent clients are a work in progress and significantly more complicated than the ephemeral case. These clients are invoked once and passed many requests through the lifetime of the client. The request format is dictated by the underlying implementation, whether FastCGI, SCGI, etc. There may be multiple simultaneous requests, depending on the implementation. Each request must be processed separately and the output handed back to the CGI server when it is completed.

The isolation model of the *kcgi* persistent client is correspondingly more complicated. Upon invocation, the web application spawns a pool of processes, sized to the maximum number of simultaneous requests, with each inheriting one readable and one writable socket. The readable sockets are maintained by the client for use in reading parsed and validated data. After the pool has been started, a control process is started that inherits the write portion of the pool's read pairs.



The control process reads requests from the CGI server, pushing data per request into the pool. When the pool parses and validate pairs, it passes them to the *kcgi* CGI client, which builds up a request object. Finally, when all pairs have been read for a given request by the *kcgi* CGI client, the request object is passed to the web application.

The web application must then pass its response to the web server directly or via the control process.

3 Implementation

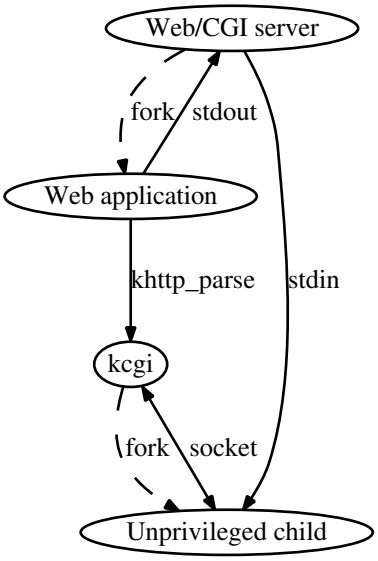
kcgi is implemented as a programming library in ISO C, libkcgi. The library consists of a single primary function that parses and optionally validates data from the CGI server, then a number of functions for writing output, configuring URLs, and so on.

Web applications invoke *kcgi* via a library call that returns a parsed and validated CGI request. It is the job of the library to properly enact its security measures with minimum involvement on the part of the web application, such that CGI, FastCGI, and SCGI types are similarly (and consistently) managed.

The source bundle consists of a portability framework for compiling and running across various popular UNIX systems, such as OpenBSD, Mac OS X, Linux, and FreeBSD. The portability framework also selects the security mechanisms available. The package also consists of libraries working atop the *kcgi* output functions for the popular HTML and JSON output formats.

3.1 Ephemeral Requests

In the ephemeral case, the web application is invoked as a CGI child of the web server process. Thus, the child inherits the security mechanisms of the parent process.



In the current *kcgi*, implementation, the `khttp_parse(3)` function is invoked by the web application as early as possible. The function then creates a socket pair and spawns a child, with an option (via the extended `khttp_parsex(3)`) for the child to scrub its parent's memory. The socket pair is one-directional, with the parent reading and the child writing. The child process inherits its parent's standard input, which it uses to read input from the CGI server.

For future work, we will allow the security policy of the persistent request to apply to the spawned children of the ephemeral case – just in case the web server is poorly configured. For example, few web servers will `chroot(2)` CGI processes by default. If this is the case, the *kcgi* spawned child should endeavour to do so.

Once spawned and before processing data, the child and parent work together (or just the child works, depending on the operating system) to sandbox the child process. To date, this uses `capsicum(4)`[6] (FreeBSD) [7], the `sandbox_init(3)` framework (Mac OSX), or `systrace(4)`[7] (OpenBSD) [4]. The protocol for sandboxing follows OpenSSH, and can be extended for other popular sandboxes.

The sandbox policy depends on the implementation, but amounts to a "pure-computation" environment with read ability on standard input and write on its half of the socket pair.

[6] `https://www.freebsd.org/cgi/man.cgi?query=capsicum&sektion=4`
[7] `http://www.citi.umich.edu/u/provos/systrace/`

3.2 Persistent Requests

Persistent requests require a more significant infrastructure due to the fact that clients are invoked apart from the webserver. Thus, the *kcgi* client itself must enact the privilege-reduced environment common in ephemeral clients. Moreover, the client must accommodate for multiple simultaneous requests through a single function call akin to `khttp_parse(3)`.

First, the process pool for handling HTTP requests is created such that the maximum number of simultaneous requests may be serviced. There must be a 1:1 relationship between parse processes and requests, since each parse process should handle at most a single request at a time. This prevents exploited parse processes from accessing other requests.

Each process drops privileges and isolates on the file-system. Following that, it (in cooperation with the parent) enacts a sandbox for the child process. The privilege dropping machinery is well-defined in `slowcgi(8)`[8] on OpenBSD; *kcgi* uses the same. Child processes are left with only a writable socket (for parsed and validated output) and a readable socket (for input data).

Following that, the client starts a control process. Like other child processes, it isolates itself. The control process is responsible for accepting connections from the web server, then passing the input descriptor into a parser and the output descriptor to the client. It manages these communications by way of socket channels between all components.

While there are several alternatives to this model, such as on-demand spawning of child processes to parse data, the complexity is not considerably changed by deviating. When a request is served by way of the web application, the parsers, client, and control process will need collectively to orchestrate a response regardless.

4 Testing and Development

Testing *kcgi* is fairly straightforward, as its components are all separable. Internally, it consists of an HTTP parser in each parsing component, then the machinery between parsers, client, and server.

To test its machinery, *kcgi* has two distinct testing frameworks.

The first of these is implemented as a library, *kcgiregress*, for custom testing by callers. The *kcgirequest* framework uses CURL[9] to pass well-formed HTTP requests to a server listening on a local socket. The local socket then acts as a simple web server, translating its requests into CGI compatible data, then passing the CGI data directly to a *kcgi* context. The *kcgiregress* caller may then inspect its parsed content and pass or fail tests.

kcgiregress is useful for testing the full chain of response. The only "artificial" part of the test is that the web server is implemented by a skeleton that does

[8] `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man8/slowcgi.8`
[9] `http://curl.haxx.se/libcurl`

little more than pass requests directly to the client. This allows for testing of the parser by way of specially-crafted POST documents.

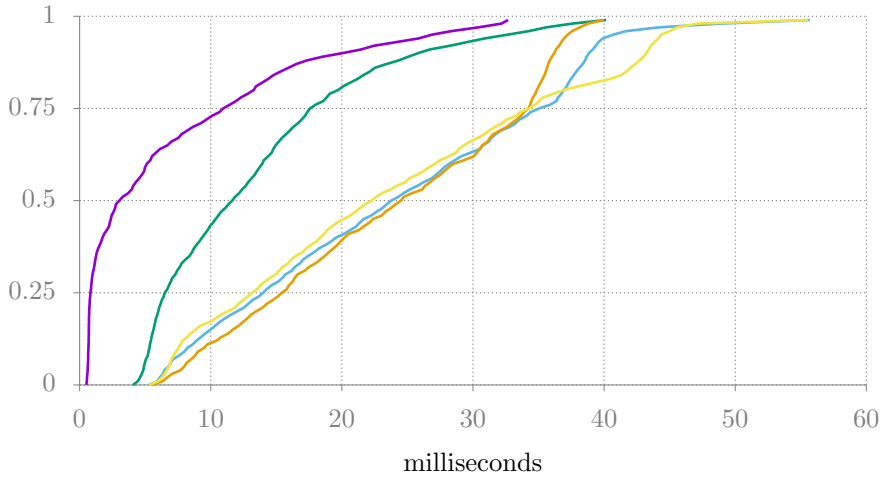
The second utility is much more focussed, designed for use with the American fuzzy lop[10]. It directly invokes the HTTP parser engine, bypassing the security mechanisms, to allow for extremely fast "fuzz" testing of the parse itself. The security mechanisms are bypassed to allow for finer-grained testing of the parser itself, along with the usual performance considerations.

The second test is useful to exercise all possible paths of the parser, which, due to the intricacies of MIME handling, are fairly complicated.

To date, only a small number of custom tests have been written for the regression framework. The AFL fuzzer, however, has been in near-continuous use, though only catching one assertion failure.

5 Performance

The *kcgi* machinery naturally introduces a performance penalty due to the need to spawn children and pass messages for each process. In general, the cost to ephemeral CGI processing should be at at least twice that of a base-bones invocation owing to the secondary passing of data between the untrusted child process.



In this graph, we show the processing time CDF of 1000 sequential accesses to a stock Mac OS X "Lion" Apache server, both as a static file (in red) and stub CGI script (in green). The mean processing time for the former, our baseline, is 5.207 ms; the mean processing time for the latter, which demonstrates the overhead in spawning and communicating with a child CGI process, is 13.8 ms–over two times the baseline.

[10] `http://lcamtuf.coredump.cx/afl/`

```
int main(void) {
{
  puts("Status: 200\r");
  puts("\r");
  puts("Hello, world.");
  return(EXIT_SUCCESS);
}
```

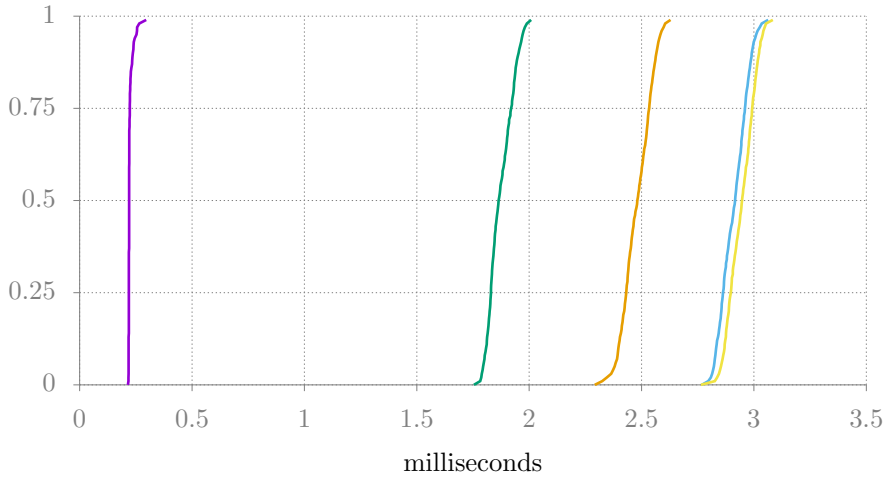
The bare CGI process is useful because static files are easily cached, which prevents a meaningful reading. We then extend the bare CGI process to invoke *kcgi* with no arguments and no option parsing.

```
int main(void) {
  struct kreq r;
  const char *page = "index";
  if (KCGI_OK != khttp_parse(&r, NULL, 0, &page, 1, 0))
    return(EXIT_FAILURE);
  khttp_head(&r, kresps[KRESP_STATUS],
    "%s", khttps[KHTTP_200]);
  khttp_head(&r, kresps[KRESP_CONTENT_TYPE],
    "%s", kmimetypes[r.mime]);
  khttp_body(&r);
  khttp_puts(&r, "Hello, world.");
  khttp_free(&r);
  return(EXIT_SUCCESS);
}
```

For comparison, we execute the *kcgi* instance in three distinct ways: the default "sandboxed" method with compression support (blue), un-sandboxed with compression (violet), and the default method without compression (light blue). It's clear that all of these are within error tolerance of the base *kcgi* case, whose mean sits at 24.5 ms–twice the unsecured CGI performance.

From this analysis, we can observe that the majority overhead in *kcgi* is involved in managing and communicating with the forked child server: no parsing occurs in this example.

Next, we use an OpenBSD 5.5 machine configured with nginx and the `slowcgi(8)` wrapper. The results are very different from the Mac OS X and stock Apache test.



In this graph, we see foremost a tremendous increase in speed over the Mac OS X and Apache instance, with the mean bare-bones CGI time being 1.913 ms and the *kcgi* being 2.954 ms, with a baseline response being 0.259 ms. Second, we see much more of an impact when running without the sandbox mechanism (`systrace(4)` on OpenBSD) at 2.52 ms. This indicates that the `systrace(4)` sandbox method has more setup overhead, which makes sense given that the `systrace(4)` method requires each system call to be ignored or discarded via `ioctl(2)`.

These performance metrics more or less illustrate the hypothesised penalty. Moreover, until a meaningful FastCGI implementation is available, these will remain: the overheads of process spawning, interprocess communication, and security sandboxing are necessary. A small amount of overhead processing may be reduced (e.g., by not having a nested loop while setting the per-system call `ioctl(2)` initialising the sandbox), but very little.

6 Conclusion

The field of web application security is, to date, exclusively in the hands of web application developers—not those writing the CGI client tools themselves. With *kcgi*, the transport security is absorbed into the client itself, allowing web application developers a consistent contract with their validated data.

References

[1] Ben Bullock. Comparison of CGI libraries in C. `http://www.lemoda.net/c/cgi-libraries/`, June 2014.

[2] Éric Faurot. OpenSMTPD. `https://www.opensmtpd.org/presentations/asiabsdcon2013-smtpd`, 2013. AsiaBSDCon.

[3] Damien Miller. sandbox pre-auth privsep child. `https://lists.mindrot.org/pipermail/openssh-unix-dev/2011-June/029657.html`, 2011.

[4] Niels Provos. In *In Proceedings of the 12th USENIX Security Symposium*, August 2003.

[5] Niels Provos. Privilege Separated openSSH. `http://www.citi.umich.edu/u/provos/ssh/privsep.html`, 2003.

[6] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, August 2003.

[7] R. N. M. Watson, J. Anderson, B. Laurie, , and K Kennaway. Capsicum: practical capabilities for UNIX. In *In Proceedings of the 19th USENIX Security Symposium*, August 2010.

A CGI Libraries Examined

The following is a list of libraries examined to establish baseline operation. For a list of C libraries, see [1].

CGI(3p) The popular way of parsing CGI forms for the Perl programming language.

cgi.py Like CGI(3p) but for the Python programming language.

cgic This seems to be the most popular CGI programming library. It was the original choice of a parser for *kcgi*, but its license prevents its use.