Red7 Communications, Inc.
PO Box 27591
San Francisco CA 94127-0591
USA

Online at:
www.cyberspark.net

To: CyberSpark Open Source Project

From: Sky

Date: April 8, 2013

Subject: Documentation of **CyberSpark** the *service/daemon,* the *sniffers, filters*, *tripwires* and *utilities*.

This document: was designed to be usable in OpenOffice, LibreOffice, Microsoft Word and other software capable of using these formats. Only the DOC version is made available online, and we check it to be sure it's at least readable in the other programs.

Table of Contents

Although the bulk of the code was written by myself, I thank John D. Tangney, who has assisted in code reviews and added some ideas to the project. And the code is in use at one other installation as well as at our CyberSpark project, so we know it's possible to make it work outside our limited testbed. —"Sky"

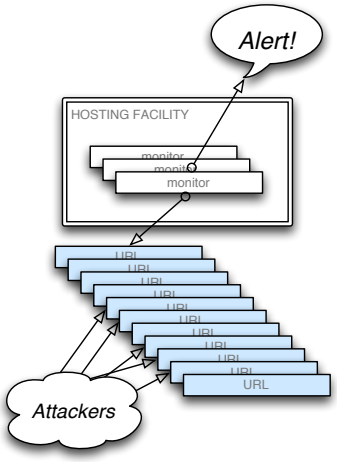
The Monitoring System

The **CyberSpark** monitoring system provides a platform for alerting bloggers, journalists and human rights NGOs when their web sites come under attack. This important task is accomplished on cloud servers running this software at several hosting facilities around the world.

The software is free and open source, written in the PHP scripting/programming language. This document describes the system architecture and focuses on operational aspects of the programs that conduct the actual monitoring or *sniffing*.

The project's software evolved from early efforts as early as 1999. Initially written in Perl and intended to look for marketing information, by 2003 it had turned into a private search engine and knowledgebase written in Java. The software was rewritten in PHP in 2010 to provide greater portability and make it easier to find programmers capable of modifying or adding to the project.

The grand plan was originally to provide a self-service monitoring system where individuals or organizations could sign up, then configure and maintain their own monitoring targets. The core of that plan, of course, was the monitoring operation itself, which is what is currently in operation.



This diagram illustrates the core concept. At a hosting facility we run *monitors* (now called *sniffers*) that reach out and examine URLs on vulnerable sites (the blue boxes in the diagram). Attackers (in the *cloud*) may disrupt those services or inject malware, and our monitoring software is designed to detect this and send *alerts* via email or SMS to parties who can respond to the issues. We call this *external monitoring*. We also have *tripwire* software that can be installed on targeted servers — this software examines the server "from the inside" and detects many types of disruptions that our outside service can't see. The tripwire software must be voluntarily installed and then "connected up" to our monitoring services to be fully effective.

Infrastructure

The monitors run on Ubuntu (Linux) or other operating systems on *virtual servers*. They're installed in cloud facilities such as Rackspace Cloud or Amazon EC2. Individuals could implement them on any kind of computer or server, but we develop on and use Ubuntu Linux. They require only PHP 5.x (which must be runnable at the command line) and a couple of PHP-based support packages. They do send email, and an external SSL-capable email server is required for alerting. They are designed to be stealthy and hardened. Only one service is configured and responds to the outside world. Occasionally, the software needs to

connect to the outside world to conduct monitoring or “sniffing.” URL sniffing is done using HTTP and HTTPS on ports 80 and 443. Alerting is done by sending email through SMTP servers — for example, gmail. The outbound email connections are limited to SMTP on ports 465 (for SSL encrypted) or 25 (for unencrypted connections). The systems are "hardened" in the sense that they present no responsive face to the outside world except for their SSH interface on port 22. (In an appropriate virtualization environment this could also be turned off if direct “console” access is available.)There is no web server, and email or other service(s) are not required to run or be exposed. The service can be very stealthy, as it looks pretty much like an idle server running only SSH.

We run our sniffers as virtual private servers on Rackspace.com and we use the smallest available such VPS in each case. They have a very light footprint.

Installation of a new CyberSpark instance requires about 30 minutes from scratch, or about 5 minutes if performed by cloning an existing instance of a monitoring server. Rackspace, Amazon Web Services and other hosts provide the ability to take a snapshot of a running instance and store it away for later instantiation. That’s the easiest way to launch a new sniffer.

Server Requirements

The system is designed to run on Linux, and more specifically was developed on Ubuntu 8.04 LTS through Ubuntu 12.04 LTS. Since it's basically in PHP, it should run on other operating systems with minor modifications.

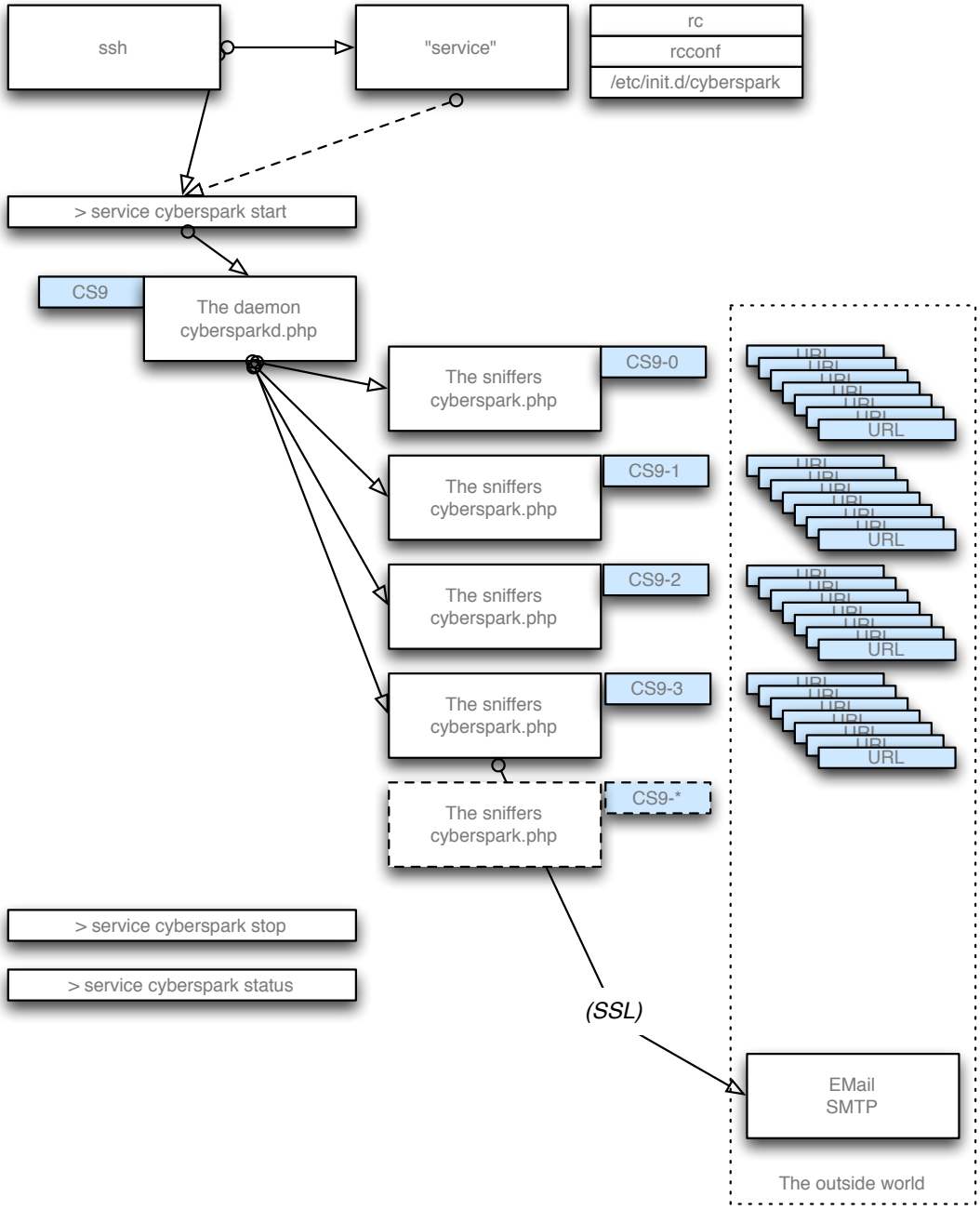
Requirements: A server with PHP 5 (5.3.2 or later recommended); PHP CLI (command line interface to start/stop PHP); PEAR Mail, Network, Sockets; a separate SMTP email server (preferably using SSL) on another server. Optional components: Google Safe Browsing free-standing server (in Python, available from Google Code) may be utilized for GSB inquiries if it's running on a separate server.

Implementing the standalone GSB requires quite a few steps, and we hope to document this at a later time. We do have a running instance that we can share with other projects that are willing to share the cost. If, however, you can bring up a virtual host of your choice that runs PHP from a CLI, you’re there.

Honestly, we implement, test and deploy on Ubuntu, and have used 8.04 LTS, 10.04 LTS and 12.04 LTS, all of which work. We previously used Debian. In theory everything should work on any system with PHP 5.3.2 or later, but we can’t vouch for this from personal experience.

Cyberspark—overview

Here's an overview of the (software) structure of a CyberSpark monitoring server. It is based on a *daemon* that is launched as the server starts up. The daemon launches processes (*sniffers*) that then follow instructions and monitor external URLs. The system below uses "CS9" as its ID, for illustration only.



Cybersparkd—the *service*

The system is designed to run on Linux, and more specifically was developed on Ubuntu 8.04 LTS through Ubuntu 12.04 LTS. Since it's basically in PHP, it should run on other operating systems with minor modifications.

Cybersparkd - cyberspark is usually installed as a ***service*** to be launched upon system startup. A file /etc/init.d/cyberspark controls the starting and stopping of the cyberspark sniffers. This is a standard method used in Debian and Ubuntu to launch and control services based on system operation *levels*. A primary process "cybersparkd.php" is launched at system startup — it looks for an available configuration file of its own, as well as for the ***properties*** files that specify which URLs are to be monitored and the conditions under which exceptions are to be reported to third parties by email. It launches one instance of cyberspark.php (note no "d" at the end) for each properties file it finds. These run as child processes, continually monitoring their assigned targets (URLs) and issuing email alerts as required. When cybersparkd is terminated (SIGINT or SIGKILL), it terminates those child processes, each of which shuts down gracefully, saving data as required, in case they are later restarted. Cybersparkd may run for weeks or months at a time, and a monitoring server could run without human intervention or interruption for years.

See notes in another section, **Staying Alive**, where there's a discussion of how the supervisory process cybersparkd monitors and controls the child processes.

Cyberspark.php—the *sniffers*

The system is layered and composed of cybersparkd.php which is the top-level service (running as a daemon), and cyberspark.php "sniffers" that it launches to do the actual work of monitoring external URLs. Below that, "filters" are the lowest level, and they do the dirty work of calculating, comparing, and watching for significant signals of change, attack or malware.

Cyberspark.php: These are the child processes launched by cybersparkd. Each cyberspark sniffer initializes by reading any persistent data from its store (on disk), then reads its properties file and begins processing directives one by one. Like its parent, it generally runs in a *daemon* mode, meaning that it will loop until terminated. It can run in a once-only mode where it drops out after processing all directives once only — this mode is primarily used when testing new code. If started by **cybersparkd** as part of the service launch process, it generally runs in this daemon mode until the master (parent) process terminates it with a SIGINT. Each "URL=" specification in a properties file tells the sniffer to spider a URL, process it using any filters that have been selected for that URL, report out by email as directed by those filters, and then go to the next directive in the properties file. Each line in the properties file is a standalone directive. The frequency of looping through the directives is also specified within the properties file. The process sleeps (yields the CPU and other resources) until it's time to sniff again.

Concurrency and threading: Because the service launches a child process for each properties file, in a system with multiple CPUs it is possible for several sniffers to be executing concurrently. Since they do not communicate with each other and share no resources, this should not pose any problems.

Known issues: We are aware that sometimes a sniffer will stall. In this state it uses no CPU time, but it is unresponsive and stops monitoring. (We are not at all clear what it is actually doing or why it stalls.) Cybersparkd will detect that a child process has stalled, will send email alerts to an administrator, and after a time it will terminate the stalled process and restart a new process using the same properties file.

"Properties" files—how to specify and tune the sniffing

*Within each CyberSpark server, it is the **properties** files that determine which URLs are going to get examined and what is to be done with each of them.*

Within the /usr/local/cyberspark/properties/ directory are any number of *properties* files. For each one of these files, cybersparkd launches a cyberspark.php *sniffer* process. The launch takes place when the CyberSpark service is started, so the actual number of sniffers on a server is determined at that time, but the URLs to be monitored, and many other parameters can be changed within the properties file while the system is in operation.

There are basically two sections to a properties file (though everything can be intermixed if desired). There are overall variables that determine loop timing, notification hour, verbosity of error messages and email. And there are "URL=" directives that cause the sniffer to go look at the URLs it is monitoring.

The *properties* file is read by the sniffer each time around its main loop (the default is every 30 minutes), and the variables are initialized each time around.

General properties (with some possible defaults or values):

```
smtpport=465
smtpserver=ssl://mail.example.com
user=abc@example.com
pass=impossiblycomplexpassword
```

These specify the SMTP port, server, username and password to be used when sending email alerts. Note that SSL is going to be the most secure method, though you can use an unencrypted connection if you wish. If you specify "ssl://" then port 465 is forced. When using unencrypted, include only the server name, not the "ssl://" – for example *smtpserver=mail.example.com*. Because each *properties* file has its own settings, each sniffer can use different email systems. These settings will apply to all URLs monitored by this particular sniffer.

```
Host=CS9 [0.0.0.0]
```

This is a plain-text "host" name that will appear in email messages.

```
Verbose=false
```

If you are debugging and running cyberspark.php from the command line, setting verbose=true will cause more messages to appear on stdout.

```
Slow=15
```

This is the time in seconds after which you want to declare an HTTP operation to be "slow." It affects the *basic* filter, which looks at slowness and nonresponsiveness for all URLs. Emails will indicate that the connection is "slow."

```
Timeout=60
```
This is the time in seconds after which you want the HTTP operation to bail out and return control. Email messages will be sent indicating that the connection has "failed."

```
To=abc@yourdomain.here
```
The default address for email notifications to a system administrator. Each individual URL being monitored can, of course, have notices sent to other addresses.

```
Pager=0000000000@txt.att.net
```
The address for "SMS-style" notifications. This is pretty much not used any more since we are sending pretty long messages. The address must be an "email-to-SMS-gateway" as we send an email, not an actual SMS.

```
From=from@yourdomain.here
```
This is the "from" address for notifications. It goes into the "From:" field of the email, and is the "purported" sender, not the actual sender. The actual sender is given in the user= field (above).

```
Subject=Monitoring alert
```
This is the beginning of the subject line for email alerts. The time and a priority indication will be added following this in the subject line of each alert.

```
Message=CS9-0
```
This is the beginning of the body of the email alert. Details are appended following this starter.

```
Load=2
Disk=85
```
LOAD= is the LOADAVG above which you want to consider the server to be slow and to "warn" you. At 2x this load it will be considered "critical." This variable can be used by any filter, but the only one using it by default is the **cyberscan** filter. Note that on servers with multiple CPUs, the LOADAVG can go higher than on single-CPU servers without causing distress.

DISK= is the percentage of disk utilization over which you wish to be notified. This is the "warning" percentage. Halfway between this and 100% is considered the "critical" percentage. As with load= this is only used by the **cyberscan** filter.

```
Time=30
```
This is the number of minutes desired between scans of a URL. Scanning time is unpredictable, and each sniffer is scanning a series of URLs, and the individual URL response times are additive. If a series takes particularly long to scan due to some temporary condition (such as a DDoS attack that lengthens the response times of a number of URLs in the series), then the timing may be disrupted. If there is sufficient time to scan all the URLs in one cycle, the sniffer will sleep until the desired time has elapsed before it begins the next cycle. But if a cycle runs long, then the next cycle will begin immediately. If you are anticipating slow sites or want to be prepared for long delays, you should adopt a strategy of putting only a few URLs in a single sniffer properties file, or you should increase the sniffer's scan time. Since there is no theoretical limit on the number of sniffers, you can always add properties files and restart the cyberspark service to start more sniffers. There are several technical factors affecting scan time and sniffer capacity, but in general you should assume a URL could take up to 60 seconds to respond (or time out) and thus you should add a full minute or more for each URL you're going to put in a properties file.

```
Notify=23
```
This is the 24-hour clock hour in which you want the sniffer to send a notification to its sysop saying that it is "still alive." "23" means 11pm server time. The server time is set in the Linux operating system. Note that this time can be set for each sniffer and is independent of the

```
Rotate
Sendlogs=email@example.com
```
During the hour when the sniffer sends its notifications (see above) it can also rotate its log file and send the log and a copy of its properties file to an administrator. The keyword “rotate” causes this to happen. If “sendlogs” is present, it specifies an email address to which the properties and log will be sent. The properties file is sent as a text file attached to the message. The log file is first gzipped, then sent as an email attachment. This zipped log file is retained on the server. A new log file is started. Since this will happen once a day, when log rotation is turned on there will be daily gz log files retained on the server, one for each sniffer that uses this option.

```
Useragent=Mozilla/5.0 (compatible; MSIE 8.0; CyberSpark
http://cyberspark.net/agent;) Ubuntu/12.04 LTS
```
If you wish to specify your own User-Agent to be presented to your monitored sites, you can add this variable. User your own HTTP:// string in order to send people to a page (which you must write) explaining who you are and why you are sniffing their sites.

You can add more parameters whenever you need them for new filters you write. The syntax is:

```
name=value
```
As long as you use a parameter name not already in use, these names and values are passed to your custom filter(s) when they are invoked. (More detail later on.)

URL properties: For each URL you want a sniffer to scan, you insert a single line in the *properties* file. This line specifies the URL to be scanned, the filters you wish to apply, and the email addresses to which you want notifications sent.

Each line consists of three sections. The first is the key “url=” followed by the URL you wish to get, followed by “;” followed by the filter names to be applied, followed by “=” and then a comma-separated list of emails addresses to be notified.

For example:

```
url=http://cyberspark.net/;dns,gsb,smartscan=abc@a.com,def@b.com
```
Either “http:” or “https:” may be used. The URL may be any valid URL. You may include HTTP name:password if you wish (http://name:password@abc.yourdomain.com) and if they are required to get you into the site. This is for HTTP authentication only – not for web apps that use <FORM> to pick up login information.

Predefined strings for substitution within Properties files

Within a properties file you may have the desire to substitute the “ID” (AKA *name*) of the sniffer within say a message, or you may need to insert the operating system type, within say a UserAgent string. You can do this by inserting:

{ID}

This inserts the ID from the "—id" parameter in the command line launch of the CyberSpark sniffer. So, for instance, if this is sniffer CS9-0, then {ID} will insert "CS9-0" wherever you put it in the properties file. For example:

```
host={ID}
message={ID}
```

{uname}

This inserts the operating system name+version from file /etc/issue.net into the line. So, for instance, if you're running on "Ubuntu 12.04 LTS" then {uname} will be replaced by that string. For example (this example would be all on one line in the properties file, please):

```
useragent=Mozilla/5.0 (compatible; MSIE 8.0; CyberSpark
http://cyberspark.net/agent;) {uname}
```

and the UserAgent used in the HTTP GET (or POST) would actually be:

```
useragent=Mozilla/5.0 (compatible; MSIE 8.0; CyberSpark
http://cyberspark.net/agent;) Ubuntu 12.04 LTS
```

The value for {uname} is copied directly from the file `/etc/issue.net` which is present in recent Ubuntu distributions.

The mechanisms that perform this substitution have been designed so that if you want to create additional substitutions, you can define them within cyberspark.php and they will automatically be available within any properties file. This substitution mechanism does not preclude the use of "{" or "}" anywhere else in the properties file.

{location}

Inserts the location, as asserted by the PHP-defined constant INSTANCE_LOCATION within ooooo. If the constant is not defined, an empty string will be substituted. For example, you might use "[London]" to indicate the sniffer is in a London datacenter. (Obviously, an interested party can always traceroute the originating IP of the sniffer to figure out where it is…)

As an example, you could put the sniffer's ID and location into the first line of all alerts by specifying "message" in a properties file. The examples below, if all were used, are probably overkill, but should give you some ideas:

```
host={ID}[{location}]
# Who the reports are to be from
from="CyberSpark {ID} {location}"<cs3@cyberspark.net>
# Default subject and message
subject=Monitoring from {location}
message={ID} [{location}]
```

The filters we supply

Here are the *filters* that are supplied by the CyberSpark project.

Basic

This filter is always applied and you can't exclude it. It looks at connectivity, slowness of response, and redirects and other simple HTTP issues.

IfExists

This is a built-in filter not present in the /filter directory. The URL is monitored and everything is considered to be "OK" as long as the URL does not respond. (404, 30x, 500 errors are all considered OK.) But as soon as an HTTP 200 response (a "normal" result) comes through, you are alerted. This can be used to watch for pages or sites that should not be available, and you'll be alerted when they appear.

Checklength or **length**

This checks the length of the returned page from the URL and notifies you if it changes. The filter remembers the lengths it has seen and only notifies you when it sees a length it has never seen before. This is primarily useful when checking JS and other script files that should never change length unless they've been hacked.

Cyberscan

Checks the results of a CyberSpark *tripwire*. (Described elsewhere.) Things like file changes, LOADAVG, and disk fullness are checked. You must have installed one of the internal tripwires at the URL you are checking.

DNS

Checks for changes in DNS (such as the DNS being *hijacked*). There's a lot that is checked. It looks for changes in "NS" in "MX" in "A" and "AAAA" records, and in "SPF" as well as changes in the SOA configuration for the domain. Only the domain is checked — the rest of the URL is irrelevant. Only certain information is checked (and retained for comparison over time) for the specific records types that are examined. As of 2013-01-05 this filter checks SOA, MX, NS, TXT, A, and AAAA. Checking for additional types depends on their being defined in the PHP dns_get_record() function.

GSB and **GSB2** and **GSBdaily**

These check your URL against the Google Safe Browsing lists to be sure you haven't been marked as hosting malware. "GSB" does this each time the filter runs. "GSBDaily" also does an extensive check where it looks at your URL, all URLs that are linked to, and all URLs linked from those pages as well. This is a powerful malware-injection checking tool. "GSB2" is not used in practice and is a leftover from some old code we hope to clean up in the future.

Smartscan

Checks JS scripts embedded within the page and linked as files. Any time one of these changes, or their number changes, a notification is sent. This can be quite powerful for sites that basically use the same javascript all the time. Sites that dynamically change javascripts, or use them in advertising, are probably not appropriate for this filter.

SSL

This is a powerful feature that checks the SSL certificate presented by the site at the URL and looks for certain important features and possible changes. You must use https:// in the URL. It checks to be sure the

cert has a valid “CA” chain, has not changed, and is working properly. All of the other filters can be used in combination with this one on secured pages. (You can also check https:// pages without specifying SSL if you wish.

Monitoring by IP address: There are times when a sensitive site may be “in preparation” for launch, but not listed in DNS for the domain. CyberSpark provides a way to monitor such a server by IP address, and *ifexists*, as described above, can alert you when the server comes online. In addition the *dns* filter (see below) watches and alerts when new server DNS records appear. To monitor a site by IP address, simply insert the IP address in the URL to be monitored, and prefix that by the site’s *virtual host* name between brackets. The virtual host name will be used in the HTTP request that is sent to port 80 on the IP address you specify. For example:

```
url=[sub.mydomain.com]http://8.8.8.8/home.html;email=me@me.com
```

would make a request of the server at 8.8.8.8, and would request that it serve up content for “sub.mydomain.com” - the server will only present content if it has been configured to do so, but will do it regardlessof whether the name “sub.mydomain.com” exists in any DNS record for the domain.

HTTPS: Yes, you can use HTTPS for any URL that serves using HTTPS on port 443.

HTTP authentication: Include name and password the usual way you would in any URL (see example), and it will be presented to the site for verification. If it fails, you’ll get a 400-series error.

```
url=http://user:password@red7.com/home.html;email=me@me.com
```

Configuration

The system is contained in /usr/local/cyberspark/ and configures itself from two general configuration files and from ‘properties’ files for each sniffer.

Directory structure: The CyberSpark directory is installed by default at /usr/local/cyberspark/

It contains these subdirectories:

/data — which contains the *persistent storage* for the system’s sniffers. The files are serialized versions of the PHP associative arrays used by each sniffer…one file per sniffer. Names correspond to the IDs.

/filters — which contains the PHP code for the filters used by sniffers. Each filename corresponds to the name used in the properties file to invoke that filter. Extra files should not be added to this directory unless they are filters, but there would be no ill effects if some were present.

/include — contains PHP code to be included in either cybersparkd.php or cyberspark.php. These are well-compartmentalized snips of functions used for specific capabilities. For example, the file **mail.inc** holds email-related functions, and **shutdown.inc** contains code used when shutting down the sniffers. We do it this way in the interest of lumping together functions with similar goals.

/log — this is a symbolic link to /var/log/cyberspark/ where the sniffers write their logs.

/properties — is a directory that holds the *properties* files. There's one properties file per sniffer. It the presence (or not) of a properties file that determines whether a sniffer is created by cybersparkd.

Several other types of files are created in /usr/local/cyberspark/.

CS*.pid : This contains the process ID for cybersparkd (example filename CS9.pid).

CS*-*.pid : Each of these files (example CS9-0.pid) contains the process ID for one sniffer. It's written by the sniffer when it successfully launches and begins its first loop.

CS*-*.next : Each of these is a "heartbeat" file for one sniffer. As the sniffer begins one execution of its main loop, it writes the Unix system time into this file. If it were to stall, this file would stop being updated. More about this later.

CS*-*.url : There's one of these for each sniffer. The sniffer writes into it the "current" URL that it's examining. If the sniffer hangs on the HTTP request or otherwise fails, cybersparkd can report the contents of this file, thus indicating what URL was being examined during the failure or stall. In theory an HTTP request always times out, but in practice sometimes an HTTP request just never terminates, and we do not understand why.

All of these files are temporary and are deleted programmatically when cybersparkd shuts down.

Config files:

cyberspark.sysdefs.php: This file contains **define()** PHP statements for strings and variable values to be used universally through the system. They are a few general areas:

- User-Agent string to be used in HTTP requests (can be altered by properties files).
- A specification of the once-a-day notification that is performed (by email).
- Google Safe Browsing (GSB) detail other than the GSB IP address itself.
- The names of the subdirectories that will be used for properties, database, filters, logs and PID files.
- …and some general stuff.

cyberspark.config.php: This file contains definitions that are unique to your installation.

- The SMTP server name, login, port and password your system will use to send email messages.
- Default email addresses for the daemon and the sniffers if none are specified at lower levels. Usually you'll override these by defining individual notification email information within the properties file for each sniffer.
- The URL to be used to contact your GSB server (or CyberSpark's) to check URLs for GSB presence.
- A default path to the cyberspark home directory. The app itself can detect where it's running and override this.
- If you need to define strings or variable values for a filter you are designing, put them in cyberspark.config.php.

Data storage—the persistence model

Persistent storage is provided for each sniffer so it can store its results during execution and between instantiations. This storage is managed by each sniffer cyberspark.php, and made available to the filters it utilizes.

Persistent data store: Each cyberspark.php sniffer maintains its own persistent data store in a file. The store is an associative PHP array that contains a few instance-related values and separate sub-arrays for each filter (see below for more on *filters*). The sniffer itself takes care of saving each filter's data and presenting it to the filter when it's next needed. Filter writers must take on responsibility for not clobbering data belonging to other filters. If you're familiar with WordPress, this is somewhat like WP's use of a single database shared among multiple plug-ins. The data is stored in flatfile form and does not require database software.

Filters—flexible scanning and alerting

The "filters" do the work of determining whether a URL has been compromised or isn't performing quite up-to-snuff.

Filters: Self-contained PHP filtering software can be written and dropped into the /filters directory. As each cyberspark sniffer launches, it examines these filter files and gives each of them an opportunity to initialize and assert its own priority. Once installed in this fashion, each filter is activated only when a particular *URL=* directive requests it. This means that filters may be applied where desired or required on

a per-URL basis. During the development process it is even possible to write and test new filters with minimal or no disruption on a server that is conducting “live” monitoring. A filter is presented with data in an isolated environment where it receives the URL, contents of the page (or file) behind the URL, a number of other parameters or values, and the filter can "reply" by returning both a *status* and a *message* to be sent as part of an email alert as well as recorded in a log file.

Examples of filters:

Length: For web sites with static and unchanging content, the simplest filter is the *length* filter. This filter records the length of each page or file, and if it has changed since the previous check, it generates an alert. This filter is appropriate for JavaScript or CSS files for which a change in length would indicate either that the file has been hacked or was changed by the webmaster (and presumably the webmaster would know the difference). Length data is kept in this filter's private store (maintained by the sniffer but available only to the filter).

Smartscan: this filter reduces the page so it contains only the script, and iframe contents of the page. These are then compared with the previous contents, and an alert is sent if they have changed. Only the contents on the page are compared, though the filter certainly could be extended to fetch any contents from a *src=* JavaScript specification using only the tools readily available in this system.

Gsb, gsb2 and *gsbdaily*: Google Safe Browsing is a system that searches out malware in web sites and flags them. Browsers that subscribe to this service will show a warning page if you try to navigate to a site that has been flagged as containing malware. This database can be interrogated, though the API is moderately complex, and any URL can be checked against the database. Because of its complexity, we do not provide code as a part of the CyberSpark package, but instead we interrogate by making requests of a separate web-based service. (You can set up your own by downloading source from Google Code and installing it on a separate server.) the *gsb* filter checks the URL and all links contained on the page against the a gsb web service and generates an alert if any of the domains is on either the malware or the phishing lists. The *gsb2* filter checks the URL itself, any pages that page links to, and any pages those pages subsequently link to. In other words it checks "two levels" out from the base URL. Because this can generate hundreds of links to be checked, and thus hundreds of requests to the gsb web service, we have also made a *gsbdaily* filter, which only performs this exhaustive check once a day per URL. The filter is capable of surviving even if the gsb web service goes down, and can alert a supervisor by email when this happens.

dns: This filter watches you DNS entries for changes. In most domains, changes are infrequent, and could indicate a *domain hijacking*. The records watched are SOA, MX, NS, TXT (including SPF), CNAME, “A” and “AAAA”. Any change in any record is reported, including record additions and deletions.

Alerting

Alerting: All alerts are sent by email. Preferably your interface to an SMTP server is ssl-encrypted, though it may be through any SMTP server on any available port you wish to use and need not be

encrypted. If you specify port 465, a secure connection will always be attempted. We recommend, for the sake of resiliency that you use a mail service that is robust and won't fail under attack. We do not suggest sending alerts directly from the monitor server itself, though you could certainly do that if you wanted to combine monitoring with such other (not so secure) services on the same machine. Sending SMS text messages can be accomplished by sending to an email address associate with the mobile device, if such an address is provided by the mobile carrier. These addresses are typically of the form 0000000000@txt.ATT.net (where the phone number replaces the zeros, for AT&T subscribers, for example).

Inside a filter

Writing a new filter: Each filter must reside in a file with extension ".php" within the /filters subdirectory. When a sniffer launches, it checks each file in this directory, attempting to execute a set-up function with the same name as the file, and that set-up function may install any number among three types of *callback* functions. The first type, *init*, is executed by the sniffer after all filters have been installed and before the first pass through any monitoring loop. This would provide the filter with an opportunity to initialize itself, read any private data that it needs from elsewhere (either on the server's file system or on a remote URL) and set up the filter's private store. The second type is the *destroy* callback, which is executed when the sniffer enters a voluntary shutdown process (such as when a sysadmin executes a KILL –INT on the process, or uses the *service* command-line utility to stop the Cyberspark daemon). This is the filter's opportunity to save anything special or to alert a remote web service if necessary. And probably most important the *scan* callback, which can examine the contents of a URL and trigger alerts or messages. Each of these is presented with the private data store as it begins execution, and can write values into the store at any time. Those values will be preserved from one execution of the filter to the next.

To write or test a filter without upsetting a production system: Even though you generally don't want to put experimental code on a production system, it is possible to write a new filter, or to change an existing filter on a production system. The process is dangerous in that it can upset the running production system, but if you're careful you can do it properly. First, you have to have the daemon running ("/etc/init.d/cyberspark start" OR "service cyberspark start") - it will read all properties files and set up sniffers for each of them. These sniffers will continue to run while you're testing. Run *top* or *htop* to stay on top of what's running.

Now, duplicate an existing properties file, or create a new one, with a different name. (Let's say you currently are running CS8 and have CS8-0 and CS8-1 sniffers running using properties files CS8-0.properties and CS8-1.properties...you might want to create CS8-2.properties for this testing.) You may have to make a few changes within the properties file, particularly to indicate the sniffer's name is "CS8-2" rather than the one you copied. Now you can create a new filter file. Be sure it uses a name not in use already, and adjust the name of the primary set-up function within that file so it matches the file name (without the ".php" extension of course). If you're rewriting an existing filter, you copy the existing file and change the set-up function name, but you must also change the names of other functions included in the filter, otherwise they'll conflict with (duplicate) existing function names from the old filter. (If you plan to leave the old filter in place, you can always choose to use those old functions and simply not duplicate them at all in the new filter.)

In order to test the new filter, you'll need to create or alter an existing *url=* directive in the new properties file (CS8-2.properties for example). Insert the name of the new filter in the ";conditions=email@email.com" conditions section. If your new filter is named "test" then you might have a new directive like:

```
url=http://web.red7.com/;test=me@cyberspark.net
```

Make all of the coding adjustments you want to make within the new filter's PHP file. The new filter is ignored by the running sniffers because they only load filters when they start up. (If you restart cybersparkd, of course, then your new filter will be discovered, along with the properties file, and a sniffer process will be created to run it...you won't want to have this happen when you're testing.) Note that PHP-cli (command line interface) must be installed for any of this to work. You can now run a cyberspark.php process to directly test the new filter:

```
php /usr/local/cyberspark/cyberspark.php --id CS8-2
```

This does a one-time run which will help you test the filter. You can freely insert *echo* statements in the PHP code to help you debug, since you're running from the command line. The output will appear in your terminal or console session.

If you make any changes to existing filters' code, they may be picked up by the running sniffers - so you'll want to avoid that unless you're very, very careful.

Once you've finished testing your new filter, you can either use it as-is and insert the new condition name into the existing properties files, or you can change its function names and replace an existing filter. While you do this you want to have the cyberspark service all shut down.

Staying Alive

The CyberSpark system is designed to stay up and running for months at a time without human intervention. Here's how it accomplishes that on a Linux server.

When **cybersparkd** launches the individual cyberspark.php instances that do the real work, it has a number of ways it can keep track of, and can discipline, those processes.

This control mechanism depends upon certain PHP functions, such as **proc_open(),** which allow one PHP process to initiate and monitor another process. We have attempted to use the most broadly-provided functions and not to depend upon any system-specific functionality. As cybersparkd launches the child processes, it retains information about them in an array and uses this information to attempt to interrogate the child processes periodically. As it completes launching the child processes, it sends an email alert (the destination email address must be in cyberspark.config.php, which is localized for each installation). Note that as each child process starts, it's actually an "sh" (shell) process that starts, which executes a command line that causes PHP to launch the actual cyberspark.php script in each case. (You can view this hierarchy using **top** or **htop** on an operational server.) So for each sniffer there is the "sh" controller and its child the actual cyberspark.php sniffer. See the screen capture below which shows a typical hierarchy as exposed by **htop**. The order of the sniffers "CS9-0" etc. is unimportant…they are all at the same level.

```
`- /bin/sh /etc/init.d/cyberspark start
|   `- /usr/bin/php /usr/local/cyberspark/cybersparkd.php --id CS9 --daemon
|       `- sh -c php /usr/local/cyberspark//cyberspark.php --id CS9-2 --daemon
|       |   `- php /usr/local/cyberspark//cyberspark.php --id CS9-2 --daemon
|       `- sh -c php /usr/local/cyberspark//cyberspark.php --id CS9-1 --daemon
|       |   `- php /usr/local/cyberspark//cyberspark.php --id CS9-1 --daemon
|       `- sh -c php /usr/local/cyberspark//cyberspark.php --id CS9-0 --daemon
|           `- php /usr/local/cyberspark//cyberspark.php --id CS9-0 --daemon
```

Cybersparkd goes around its main loop periodically (the default is several minutes), checking the child processes, checking the child heartbeat files, updating its own heartbeat file. Once a day it also sends an email to indicate that it's still running. Since 2010, when the PHP version was written, we have not had this main loop fail (or even stall) on any of our Ubuntu servers.

When the child processes — the sniffers (like "CS9-0" above) — are launched, each one writes a PID file (upon successful launch) containing its process ID, so the parent daemon can verify the child process has launched, and can later on terminate (gracefully or forcibly) individual sniffers as needed. That's also how the graceful shutdown is accomplished when you use:

```
/etc/init.d/cyberspark stop
```

or

```
service cyberspark stop
```

This does a

```
kill –INT processID
```

on the parent process cybersparkd, which will gracefully shut down the child processes. It does this in two ways. First, it sends a SIGINT (ctrl-C equivalent) to the sniffer to shut it down gracefully. Then it also calls (PHP) **proc_terminate()** on the 'sh' that surrounds each sniffer, waiting a while for that to shut down (it uses **proc_get_status()** to check status in a loop while waiting), and then using **proc_close()** to remove the process information. This shutdown of child processes is equivalent to a ctrl-C being executed on each child when running from the command line. The child processes are equipped to catch these shutdown signals and gracefully terminate, as long as they're not stalled or crashed. If the child process terminates gracefully, it will remove its own PID file. In an emergency, a sysop can also kill –INT any of these processes and they'll terminate gracefully.

There are situations during execution where a sniffer may run into trouble and either become unresponsive, or completely shut down. If it is able to remove its PID file, then the cybersparkd daemon (the parent) will attempt to restart it later on when it discovers the PID file is missing.

But there are also situations where a sniffer may get into trouble and get hung up in a condition where it can't shut down, can't monitor, and can't respond to a kill –INT (ctrl-C equivalent). This might be due to a programming error or some unanticipated PHP or operating system problem. It might even be due to some exploit the remote site can inject into the results of the monitored URL! To guard against such silent failure, each cyberspark.php sniffer writes a heartbeat files each time it begins its main loop. The heartbeat file contains the Unix (integer) time that the monitor predicts it should "next" come around the loop. For example, if it's now 1:00pm server-time and the loop is beginning execution, and if the monitor is supposed to run every 15 minutes, then 1:15pm is the predicted next-run time. This time, as an integer ("Unix time" begins with the start of the unix epoch in 1970), is stored in the heartbeat file. The Cybersparkd daemon looks for these heartbeat files and compares their contents against the current time, and if one becomes grossly outdated, cybersparkd can decide to attempt a kill -INT against the process, or it can simply kill the process outright. (The default is to wait 30 minutes beyond the heartbeat time and then perform an outright

kill -QUIT of the malingering process.) It also deletes the corresponding PID file, so the next time cybersparkd checks for the process it and its PID will be gone and cybersparkd will restart that specific monitor.

"Hard" system restarts: On occasion a server will lose power or be forcibly shut down. In such cases, the PHP processes aren't given a chance to clean up. The /etc/init.d/cyberspark script that runs when the system reboots can detect this. The "rc" process runs:

```
service cyberspark start
```

which initially looks for the presence of *.pid files in /usr/local/cyberspark and if it finds any, it checks to see whether the corresponding processes (remember the process ID is in these files) are alive. If they aren't, it will write a message to sysout, delete the files (*.pid, *.next and *.url) and then go ahead and restart the service normally. If it finds any kind of snarl, such as perhaps cybersparkd is still running (which would indicate this is not a cold start), it will exit without starting. No email notifications are sent (good idea to do this for the future?).

Unresponsive Processes

The daemon launches the monitors (AKA *sniffers*) at startup.

A monitor process writes a file CSx-n.next (where "x" is the daemon number, like 7, 8 or 9 and "n" is the child (sniffer) process, like 0, 1, 2 etc.) with a unix time (in seconds) indicating when the process "thinks it should next wake up." In our documentation and code we call this the "heartbeat" file. A monitor that runs every 20 minutes, for instance, would write a time that is 20 minutes in the future. It writes this file each time it wakes up and begins looking at URLs. If it gets bogged down or something fails and it never completes the loop ("gets stuck") it will not update this file again.

The daemon deems a sniffer to be "unresponsive" when the sniffer's heartbeat time has passed. In other words, a monitor running every 20 minutes is deemed "unresponsive" if it has not waked up after 20 minutes and refreshed the heartbeat time. We actually cut it some slack - a minute or two, though that could be lengthened. However, being unresponsive is "relative" - we don't know whether the process is just busy waiting for remote web servers to respond, or perhaps is never going to recover.

When a sniffer is unresponsive in this fashion, the daemon notifies an administrator by email every 3 to 5 minutes (depends on a configuration setting).

When a sniffer has been unresponsive for 1800 seconds (30 minutes), the daemon will kill it with a SIGKILL. (This means the normal graceful shutdown does not occur.) We don't cut it any slack. (The reason is that there isn't a universal and reliable way in PHP to actually check the child process to determine whether it's capable of responding.)

Note: In a normal shutdown, like "/etc/init.d/cyberspark stop" from the command line, the daemon receives a SIGINT, which allows it time to gracefully shut down each sniffer, and then tidy itself up at the end. However, our monitors are actually in a "clean" state each time they finish their loop, having written all persistent information to the disk store, so an immediate and unconditional kill signal like SIGKILL wouldn't leave many loose ends (just the 'next' and the 'pid' files, actually).

The daemon comes around again about 3 minutes later to look at all of its child processes, including the one (that was) unresponsive, and notices that it is gone, at which point it restarts the appropriate sniffer.

Organization and Workflow

The cyberSpark system is designed to stay up and running for months at a time without human intervention. Here's how it accomplishes that on a Linux server.

CyberSpark was written and is maintained by "Sky" in San Francisco CA. The open source and free code is kept current on github

`https://github.com/jimsky7/CyberSpark.net`

The code is in use by at least one other group of people, and is available for use by anyone. Licensed under a Creative Commons by-na-sa license, you must:

1) give attribution to its creator "Sky@cyberspark.net, Red7 Communications, Inc."
2) use it only for non-commercial purposes; in other words you may not charge for the code or for your use of services built upon the code;
3) and you may modify and share the code as you please as long as #1 and #2 are observed. If you make modifications, you may insert your own attributions as comments within the code, and/or as github "commit" comments.

Code and documentation contributions are welcome, and please contact sky@cyberspark.net when you are ready to upload modifications or additions to the code on github. All contributed code will be reviewed before it is made public.

Tips on modifying and testing code

The cyberSpark system is designed to be a daemon in normal operation. However, when you're testing new capabilities, you will often want to run it once only, examine the output or results, and then run again.

The daemon cybersparkd.php can be run one-shot from the command line, as can the sniffers cyberspark.php. When you run in this mode, you can see debugging messages on the console, and you can even run in a "verbose" mode that will give more messages.

Let's start by looking at how to run or test an individual sniffer.

From a shell or console, and running on an experimental server rather than a production server, first shut down the cyberspark daemon:

```
service cyberspark stop
```
If cybersparkd was running, you may see one or more messages as it shuts down.

You need to have cyberspark.config.php set up with a proper ID, let's say "CS9" in this case, with email addresses and other parameters. You'll also need a properties file, let's say "CS9-0.properties" set up with at least one URL you want to test monitoring.

If you are testing a new filter, for example, set it up properly (see elsewhere) and have it ready.

Now you'll want to run a single sniffer one time only, to see whether your new filter works:

```
php /usr/local/cyberspark/cyberspark.php --id CS9-0
```
This will run the sniffer once, after which it will terminate. Note that if you use "CS9-0" as your ID parameter, this determines the properties file chosen and thus the identity of the sniffer you are testing. Messages, including errors, will be written to stdout and you should see them in your console or shell. If the code malfunctions by looping, you can shut it down with ctrl-C. To obtain maximum error messages, you may wish to turn on the *verbose* parameter in the properties file.

```
Verbose=true
```
Once you've tested single run-throughs, you can also run the sniffer as a daemon by adding one more parameter.

```
php /usr/local/cyberspark/cyberspark.php --id CS9-0 --daemon
```
If you do this, of course, you will not regain control of your shell until you either terminate the running process with ctrl-C or it crashes. If you were to close your shell, the process would also terminate. Remember that you're running only the sniffer and not the parent daemon.

To debug a filter or other new code, make your changes to the appropriate PHP filter or includes files, then run cyberspark.php from the command line. Repeat until satisfied.

It is sometimes handy to look at the log file created during debugging. This file lives within /usr/local/cyberspark/log/ for each sniffer.

Although the parent cybersparkd.php can be run directly from the command line, it's not very useful to do so. Instead, use **service** to start and stop the parent service. If you want to start cyberspark and then log out (terminate your shell) you can use **nohup** to start the service — when you then close your shell, cybersparkd and its child processes will continue to run, and it will be detached from your shell rather than terminated.

```
cd /usr/local/cyberspark/log
nohup service cyberspark start
```

CyberSpark Tripwires and Utilities

Independent of servers and sniffers, we have two PHP scripts you can install on your own server, and you can then interrogate them from a web browser (or from a CyberSpark sniffer) to learn important things about the innards of your PHP application. There's also a "Utility" that can repair certain types of code injections.

Whether you're installing external sniffers, we have two types of PHP *tripwire* scripts you might benefit from installing on your website server. In our github repository you'll find a directory "/tripwires + utilities" that contains all of the scripts and code that will be described in this section.

Cyberspark-scan.php

Independent of servers and sniffers, we have two PHP scripts you can install on your own server, and you can then interrogate them from a web browser (or from a CyberSpark sniffer) to learn important things about the innards of your PHP application. There's also a "Utility" that can repair certain types of code injections.

Whether you're installing external sniffers, we have two types of PHP *tripwire* scripts you might benefit from installing on your website server. In our github repository you'll find a directory "/tripwires + utilities" that contains all of the scripts and code that will be described in this section.

Cyberspark-scan.php can be installed at the document root of your PHP site. We call this the *tripwire* script. You may wish to change the filename before installing it.

When installing the tripwire, you must also create a directory named "cyberspark" and make it readable and writeable by the web server. (777 permission if there's any doubt) You may also wish to create server rules to hide the contents of this directory so it won't be served to any web browser. It will contain the data and logs from the tripwire's operation, and only the script itself needs to use this data. For an Apache2 webserver, for example, these configuration lines would do the trick:

```
<IfModule mod_alias.c>
# These rules may be placed within a file in /etc/apache2/conf.d/
# or in an .htaccess
# Protect CyberSpark tripwire data
    RedirectMatch 404 ^(.*)cyberspark/(.*)
</IfModule>
```

The cyberspark-scan.php script itself must be executable, and is outside this directory. This script pays special attention to PHP and JS files within the directory structure of your site. (These are the most likely to be hacked.)

You invoke this script from outside in a web browser, and you must give it certain parameters.

http://example.com/cyberspark-scan.php?params=

help This causes the script to print a short explanation of available parameters.

```
http://example.com/cyberspark-scan.php?help
```

report=n Causes the script to search "n" levels deep from the docroot looking at files and recording their lengths. If a file length changes, the script will report that change. New files, and deleted files, are also reported. The example below causes the script to search 5 subdirectories deep, starting from the docroot.

```
http://example.com/cyberspark-scan.php?report=5
```

base=/foo This sets a base for the search. It's relative to the directory the script is in. So if you have the script at docroot, but you only want to search in /wp-content/uploads you would use:

```
http://example.com/cyberspark-scan.php?report=2&base=wp-content/uploads
```

ignore=xxxxx or exclude=xxxxx or except=xxxxx When you have a need to ignore certain directories or files, you can specify them. The "ignore" string(s) will cause directories or files containing them in their names to be skipped during the analysis. If you have multiple strings, separate them by commas. If you want to skip a whole directory, you can just give the directory name, or you can put slashes before and after the name – equally effective.

```
http://example.com/cyberspark-scan.php?report=2&ignore=/cache/,git,.log
```

disk=dev or **disk=none** Specifies that you want (or not) disk stats on a particular volume. For this to work, PHP functions disk_total_space and disk_free_spacemust exist on the system. You may have to mess with this to get it reporting properly on your system. Some require the "dev/" mountpoint, and others require the actual mounted filesystem path (such as "/"). (Why would you use "none"? I don't actually recall why…) Do not use the preceding "/" on the path except in the case where you want stats on "/". So rather than "/dev/sda1" you would use "dev/sd1" instead.

```
http://example.com/cyberspark-scan.php?report=2&disk=dev/sda
http://example.com/cyberspark-scan.php?report=2&disk=/
http://example.com/cyberspark-scan.php?report=2&disk=none
```

cpu=n Calculates LOADAVG over the period of "n" seconds. Does this by artificially sleeping for "n" seconds and then reporting back the system load during that time. This adds to the time it takes your script to report out, so if you use a large number, you may find your site is reported as being slow.

```
http://example.com/cyberspark-scan.php?report=2&cpu=3&disk=dev/sda
```

Here's an example report generated by the **cpu** and **disk** options:

Load: 0.17 0.07 0.04
CPU: 2%
Checking filesystem at /dev/sda1
Disk: 0% used of 0 GB total on "/dev/sda1"

Cyberspark-utility.php

Independent of servers and sniffers, we have two PHP scripts you can install on your own server, and you can then interrogate them from a web browser (or from a CyberSpark sniffer) to learn important things about the innards of your PHP application. There's also a "Utility" that can repair certain types of code injections.

Similar to the cyberspark-scan script, this one spiders through your docroot (or other location as specified) and finds files that may have been compromised, then removes the bad code.

Many site compromises consist of adding javascript at the end of a PHP or JS file. If you find that a site has been compromised in this way, you can copy the unwanted javascript out, paste it into the file /cyberspark/removeme.txt and then run this script. The site will be spidered and the unwanted code will be removed.

Of course you want to make a full backup of your (compromised) site before you run this thing.

One additional parameter is required (over cyberspark-scan.php) which is

Repair=n Specifying that you want files replace to a depth of "n" subdirectories within your site. You can specify **base** and **exclude** and other parameters.

```
http://example.com/cyberspark-utility.php?repair=222
```