DATA EXFILTRATION PROJECT: FINAL REPORT

COMP8505/SELECTED TOPICS IN DATA COMMUNICATIONS

TORIN SANDALL

STEPHEN MAKONIN

JUNE 2008

Table of Contents

Installation

Linux Kernel Module

Unzip and build the project:

```
[root@localhost ~] tar xvf project.tar.gz
[root@localhost ~] cd project/
[root@localhost project] make
[root@localhost project] ./install.sh
```

Backdoor Application

* Installed automatically by the Linux Kernel Module.

Client Application

Unzip and build the project:

```
[root@localhost ~] tar xvf project.tar.gz
[root@localhost ~] cd project/
[root@localhost project] make
```

Usage

Linux Kernel Module

Load the module into memory using insmod:

```
[root@localhost ~] insmod bin/pcspkr.ko
```

Backdoor Application

No usage required, all input is fed from the client application and all output is sent back to the client.

Client Application

The format of the client application configuration files is documented in etc/README.

Run an instance of the client application in listener mode

```
[root@localhost project] ./bin/cli -c etc/listener.conf
```

Run an instance of the client application in request mode

```
[root@localhost project] ./bin/cli -c etc/request.conf
```

Assuming there is enough network traffic coming to or from the compromised host the command response should be received within 5-7 seconds.

Project Requirements

Since the nature of the project allows us to be very creative in terms of functionality and design we chose to create a list of interesting rootkit features which have been discussed and scattered throughout numerous articles and papers. By implementing these features we expect to gain a stronger understanding of the Linux operating system as well as methods used by attackers to steal data from compromised hosts. The features we plan to include in our system are enumerated below:

Concealment and Authentication

In order to prevent unauthorized access from external parties it is important to control who can access the backdoor process running on the compromised host. To meet this requirement we shall include timestamps and password based authentication for all command requests. In addition, the command requests shall be encrypted to prevent eavesdropping. Taking this even further, we plan to include host spoofing features to allow the client to send requests to spoofed hosts which will misdirect anyone who may be attempting to determine the source of the data leak.

Finally, by utilizing the libpcap library we are able to ignore any packets which do not meet the appropriate criteria, thereby masking the presence of the backdoor to the outside world.

Process Camouflaging

If the backdoor process was visible to users of system it would not be very useful. To prevent users from being able to discover the presence of our backdoor process we are going to be using a Linux Kernel Module (LKM) which manipulates kernel structures to hide the process, prevent the process from being killed, and even hiding the presence of the module itself. Once loaded into memory, the backdoor will effectively invisible and will not be able to be shutdown.

Surviving Reboots

We are designing the backdoor process to run on Linux workstations which we expect to be rebooted. Challenges arise in not only ensuring that our backdoor process is executed at startup but that any configuration files are not easily found. To accomplish this we will be making use of an LKM again.

Data Exfiltration

While it would be possible to simply open a connection to a remote host and echo file contents, detecting this type of activity would be extremely easy. In order to prevent other parties from detecting the data leak, we plan to disguise our data by injecting it into HTTP GET request packets destined for well known websites.

Network Usage Monitoring

Since we want to avoid being detected while data is being exfiltrated we will monitor the amount of traffic destined to or from the compromised host. In order for data to be returned to the client the workstation must be generating enough of its own traffic to pass a predefined threshold.

Keystroke Logging

To extend our systems functionality even further we plan to implement a keystroke logging feature which will be triggered by the presence of strings such as “login.php” or “hotmail.com” inside of HTTP traffic.

System Design

Data Structures

The sched_event structure can be used as a header for any other event structure.

```
struct sched_event {
	int fd;				// Descriptor to associate with epoll
	int (*callback)(void *);	// Callback function to execute after epoll event
	void (*cleanup)(void *);	// Handles memory cleanup if callback function fails
	void *ptr;			// Event dependent
};

struct file_event {
	char name[MAX_NAME_LEN];	// Name of file which will cause inotify to signal
};

struct sniff_event {
	char str[MAX_STR_LEN];		// String to match inside HTTP GET requests
};

struct cmd_pkt {
	struct ether_header eth;	// Ethernet header
	struct iphdr ip4;		// IPv4 header
	struct udphdr udp;		// UDP header
	unsigned long timestamp;	// Timestamp to prevent replays
	char passwd[8];			// Password to provide authentication
	unsigned long addr;		// Client listener address
	char opcode;			// Type of packet
	char options[0];		// Options associated with type
};

struct arp_pkt {
	struct ether_header eth;	// Ethernet header
	struct arphdr arp;		// ARP header
};

struct icmp_pkt {
	struct ether_header eth;	// Ethernet header
	struct icmphdr icmp;		// ICMPv4 header
};

struct queue_element {
	void *ptr;			// File descriptor or keystroke buffer
	unsigned long len;		// Number of bytes pointed to
	int (*callback)(void *);	// Handler function for exfiltration OR keylogging
	queue_element *nxt;		// Next queue element
};
```

Request Packet

The request packet shall be used by the client to execute commands on the compromised host. If possible, the client shall send its request packets to one of the spoofed host addresses to misdirect any parties monitoring the network traffic. The entire request packet shall be encrypted using an encryption scheme which has yet to be decided. At the moment the most logical choice would be a stream cipher such as RC4 because we want to minimize the packet length as much as possible.

Timestamp (4 bytes)	Password (8 bytes)	Type (1 byte)	Options (N bytes)

Timestamp

To be computed by the client and placed in the first 4 bytes of the request packet. The purpose of the timestamp is to prevent replay attacks from being successful. The method of generating the timestamp has yet to be determined however it must be noted that the backdoor process must check both the validity of the timestamp and whether or not the timestamp has been seen previously.

Password

To be stored in a configuration file on the compromised host and passed to the client at runtime. The purpose of the password is to prevent unauthorized users from accessing the backdoor process. It must be noted that the password must be no more than 8 bytes long and if it is under 8 bytes it must be padded with NULL bytes.

Type

A single byte op-code which specifies what type of command is being requested. At this point in time, there shall be three valid values:

1. **Exfiltration Request:** In this case, the client is requesting that a file be returned. The filename will be placed in the options field and will either be in logical form (in which case the backdoor process will check scan its directory list to see if the file exists) or in absolute form (in which case the backdoor process will simply open a file descriptor based on the path provided).

2. **Keylogging Request:** In this case, the client is requesting that the backdoor process monitor keystrokes for a preset amount of time. The options field must contain a string (such as login.php or hotmail.com) which will then be used to match against outgoing traffic. When a match is found the backdoor process will begin buffering keystrokes for the preset amount of time.

3. **Execution Request:** In this case, the client simply wants to execute a system call on the compromised host. The options field must contain a string which represents the commands to execute.

Options

Variable length field which contains data that relates to the type of packet as described above.

Exfiltration Covert Channel

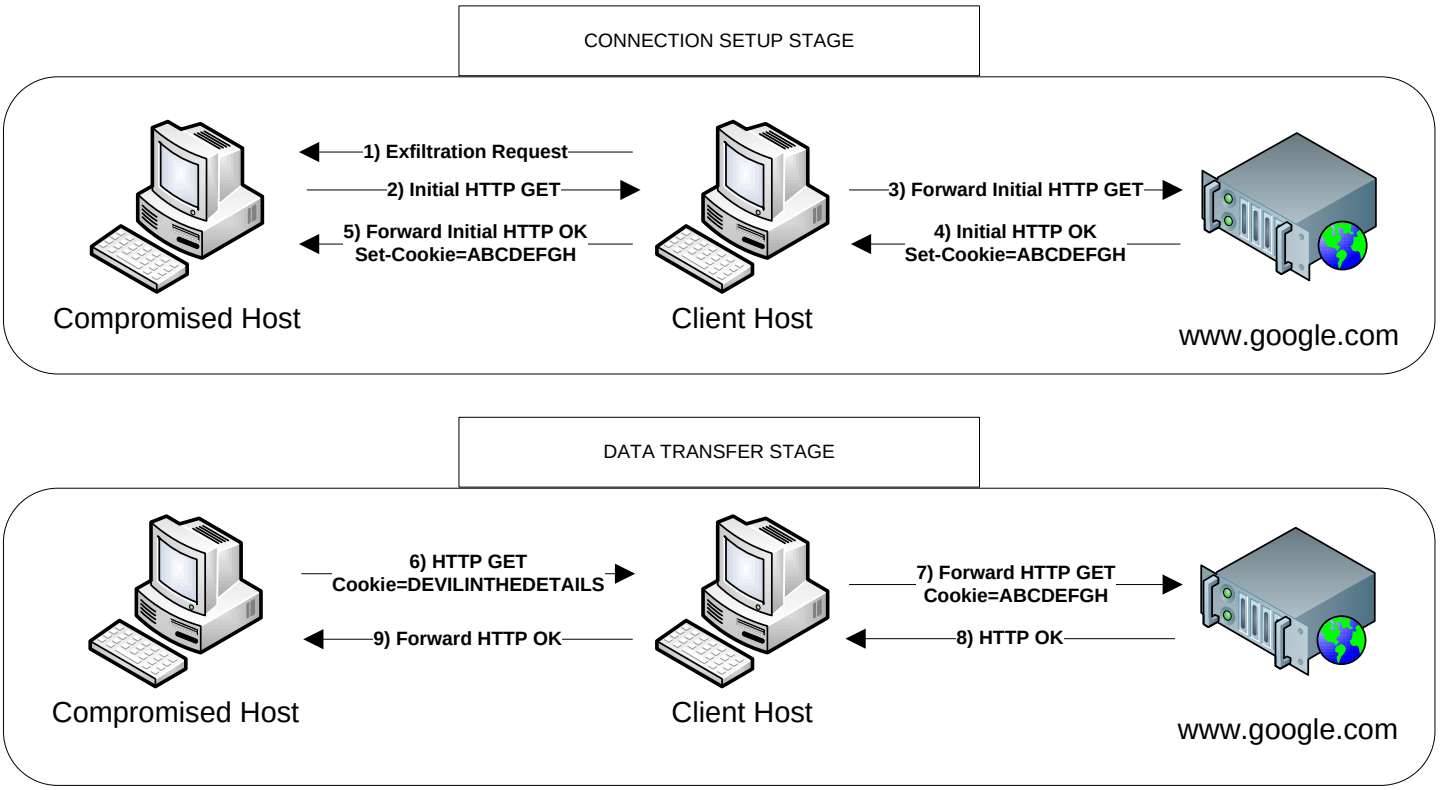
Our covert channel will take advantage of HTTP 1.1 by sending seemingly innocuous GET requests to publicly available websites. For the purpose of this project we have designed our covert channel to communicate with www.google.com. A description of our covert channel design follows.

Like many other websites, Google assigns a Cookie value to each connection which it handles. This value is useful in determining user information and maintaining state on the server's side. After the compromised host has connected back to the client and the client has setup a connection with www.google.com, the client will receive the initial HTTP GET request from the compromised host and forward it to www.google.com. At this point www.google.com will respond to the client with a HTTP OK response containing a Cookie value. The client will forward the HTTP OK response to the compromised host, which will then begin transmitting subsequent HTTP GET requests containing data encoded in the Cookie field.

While it would be possible to inject arbitrary amounts of data into each HTTP GET request this could easily trigger network IDS sensors along the way. In order to avoid this we will only inject as much data into the HTTP GET requests as the authentic cookie value would contain. We will attempt to conceal the presence of the covert channel even further by selecting keywords from files located on the compromised host when crafting our HTTP GET requests.

For each HTTP GET request which the client receives from the compromised host it will strip the encoded data and replace it with the authentic Cookie value. After the client has done this it will forward the HTTP GET request to www.google.com and will await the HTTP OK response. Once the client has received the HTTP OK response from www.google.com it will forward the response to the compromised host. This process will continue until the compromised host has no more data to send.

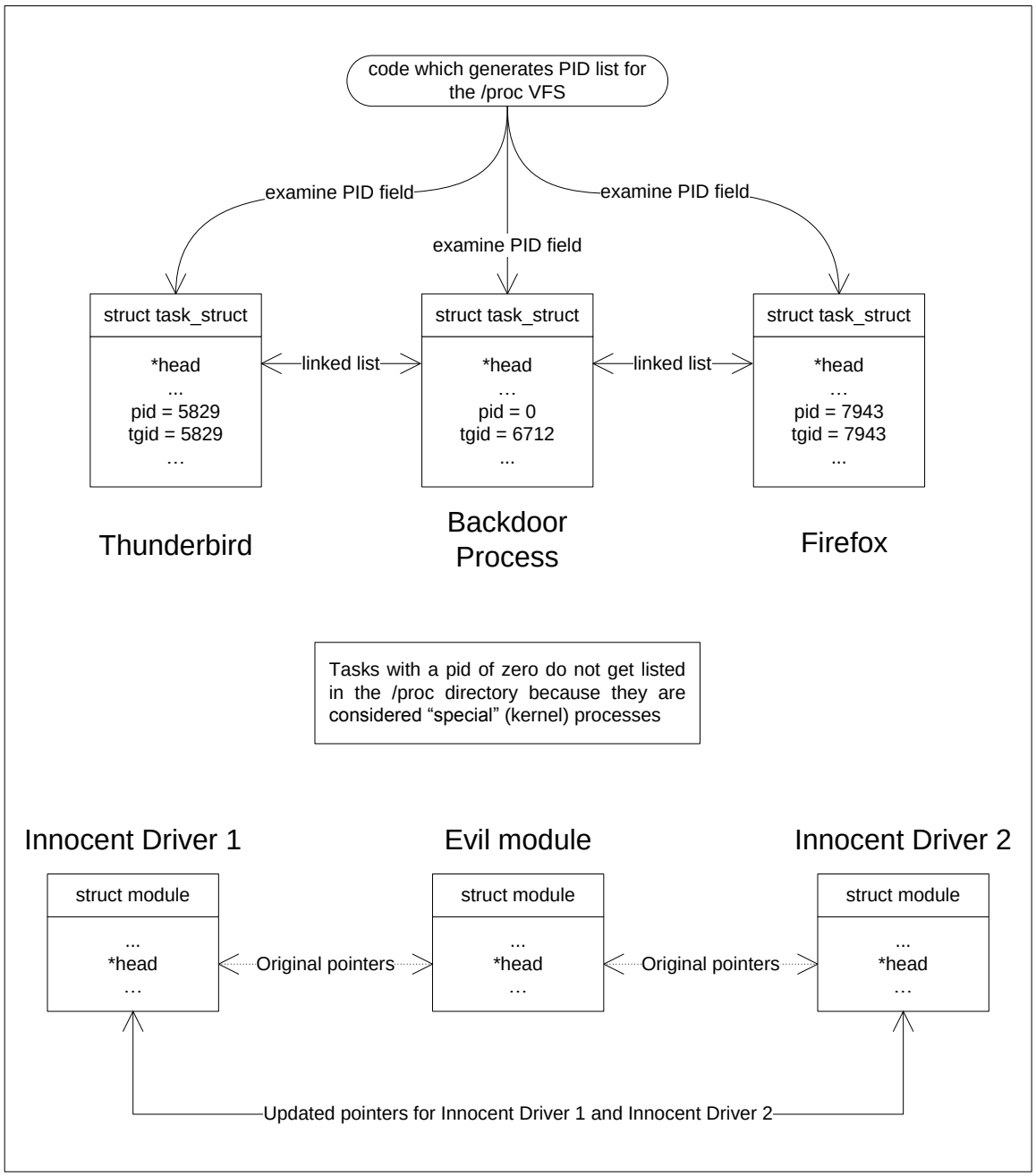
The following diagram illustrates the entire process:



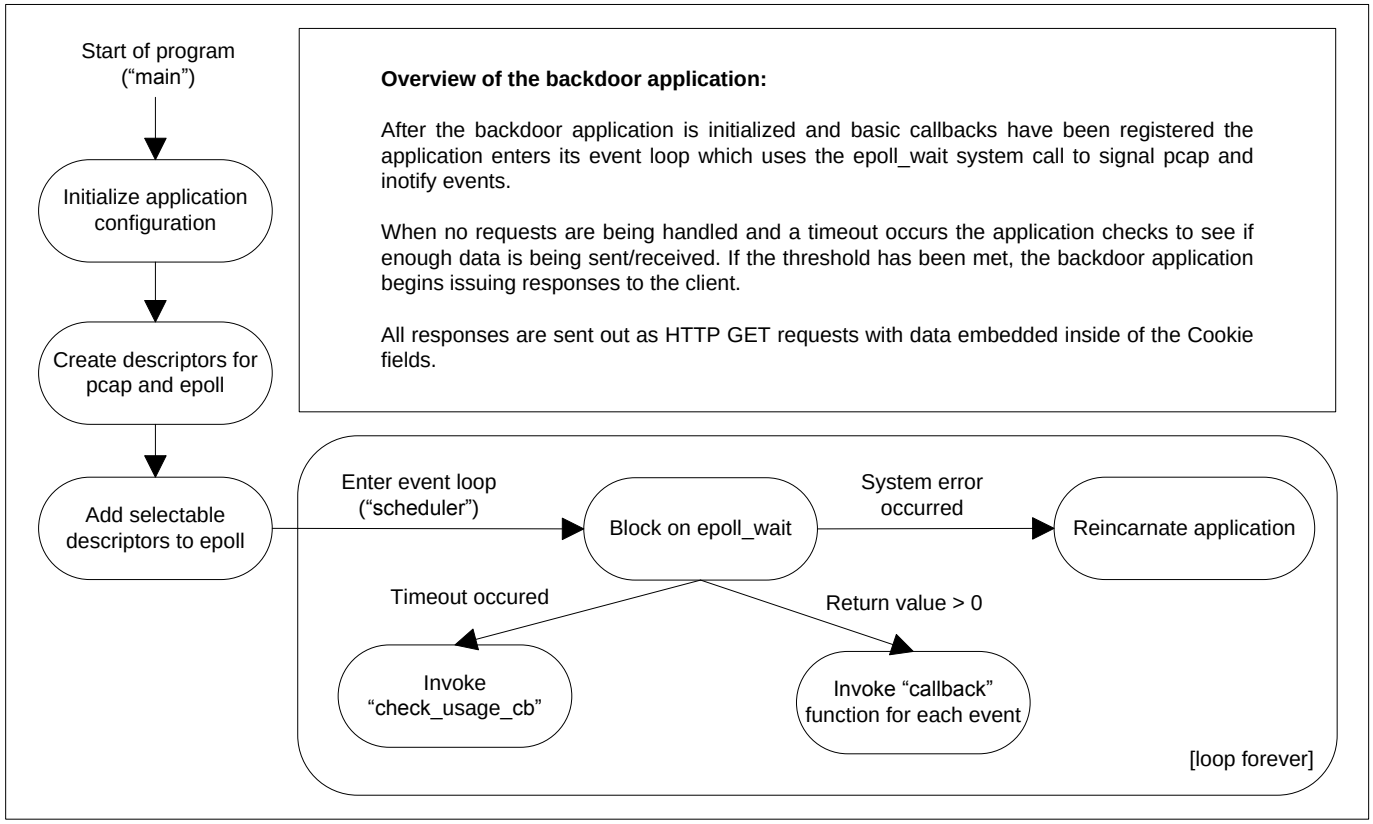
Process Camouflaging

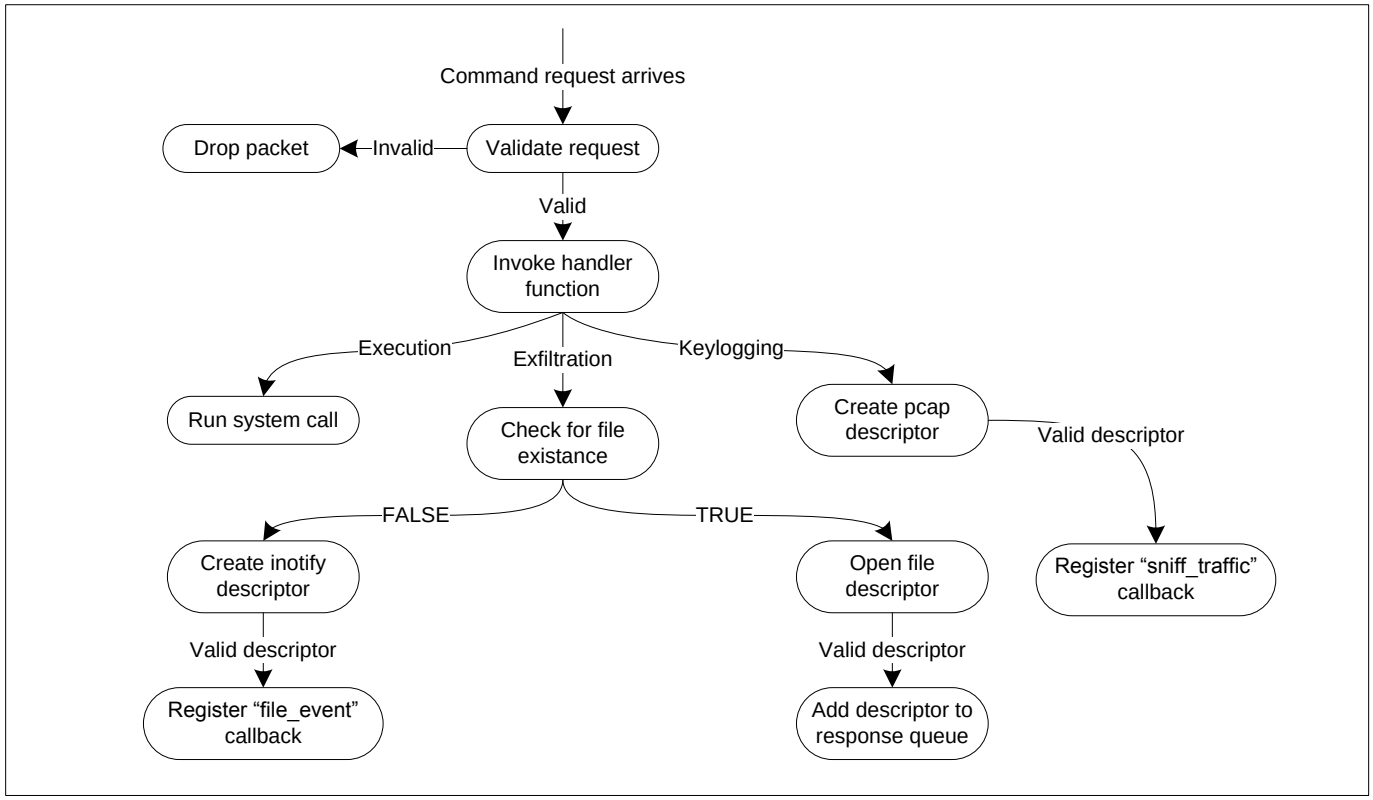
In order to hide the presence of our backdoor application we take advantage of a Linux Kernel Module which allows us to manipulate kernel structures which are used in multiple places, but most importantly in the listing of applications and modules. The method used to hide the backdoor application was outlined in Phrack Issue 63. The method works by locating the task_struct structure associated with a process and then zeroing out the PID field. This effectively prevents the process from being listed by programs such as **ps**. This method also prevents the application from being killable and guarantees that no other process will receive the same PID.

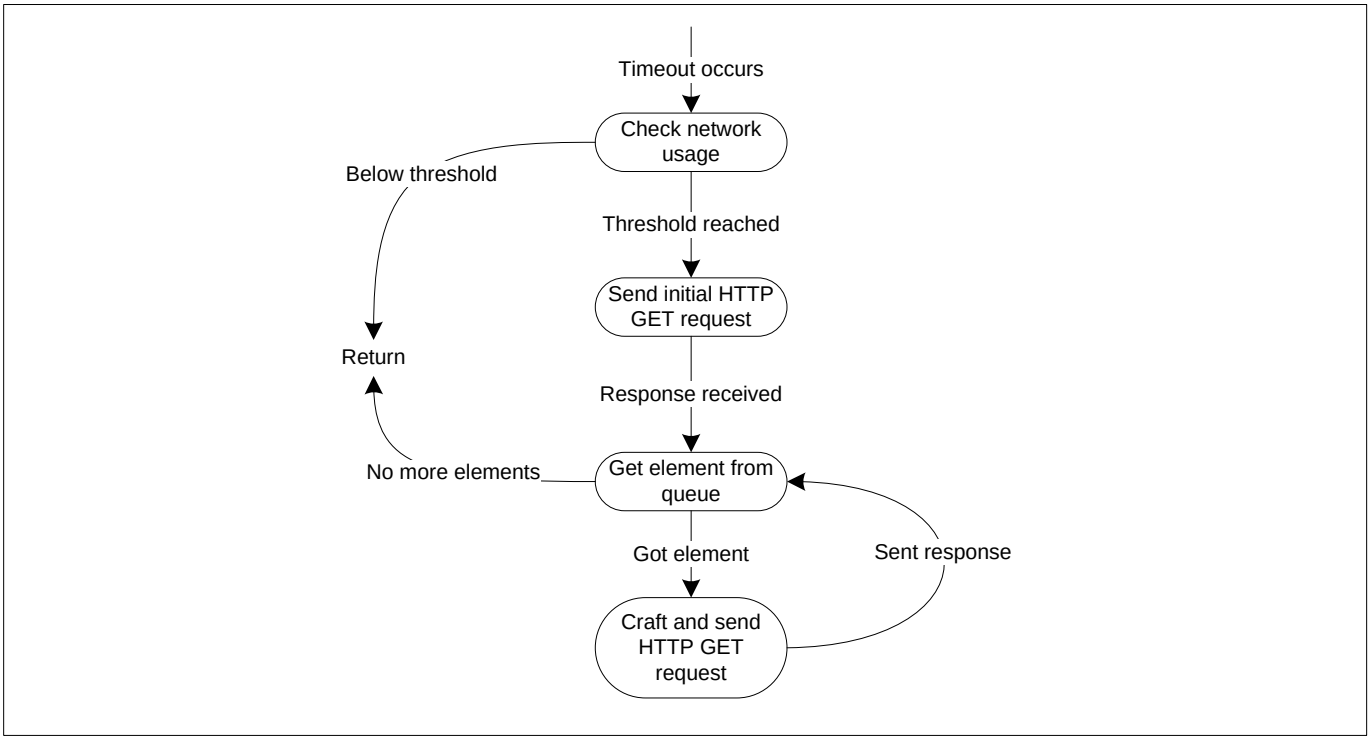
To hide the module which could otherwise be found with a simple call to **lsmod** we locate the module's structure and update the linked list pointers for the previous and next module structures. When **lsmod** lists the active modules it will simply skip over our module. The following diagram illustrates the entire process:

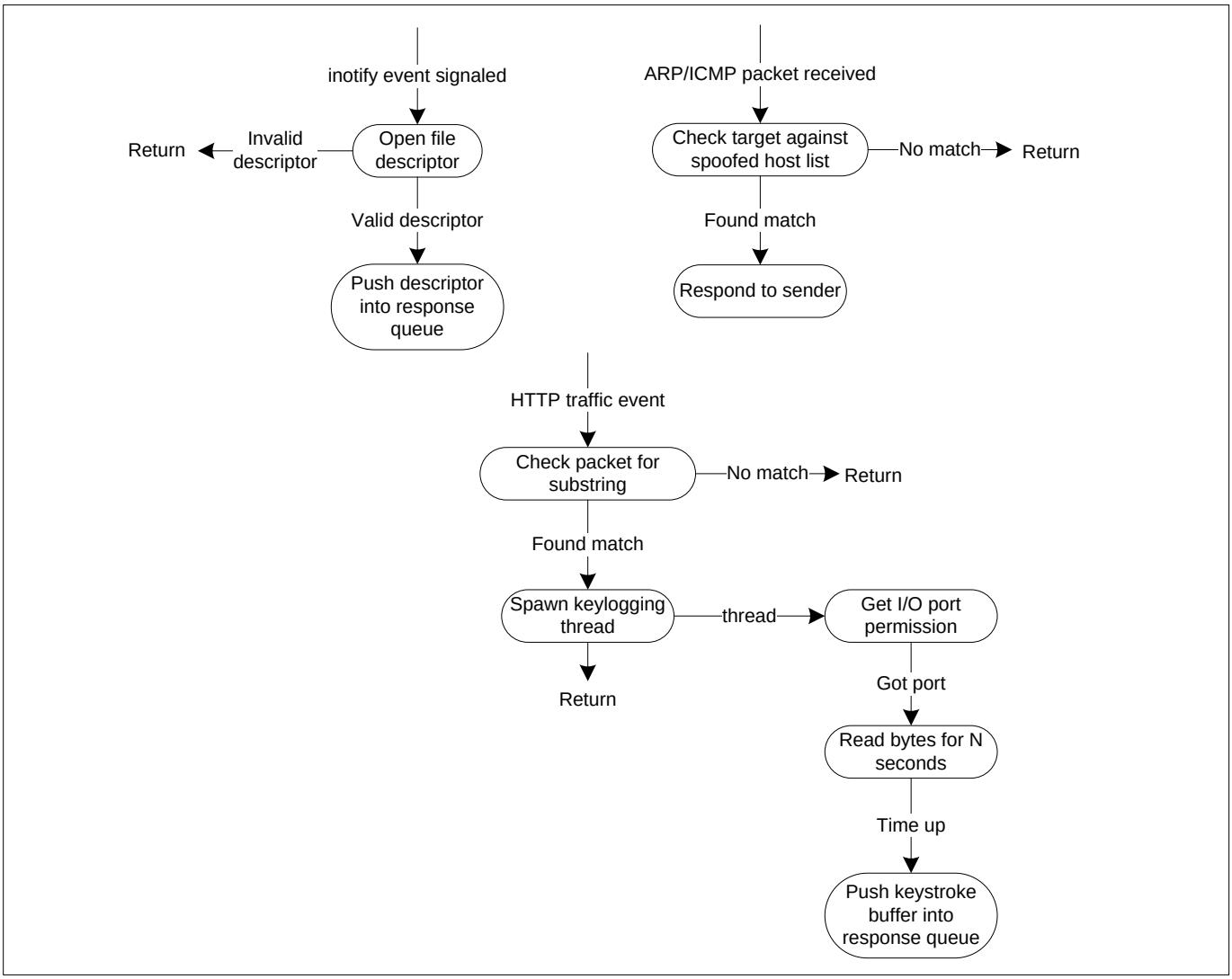


Server Architecture

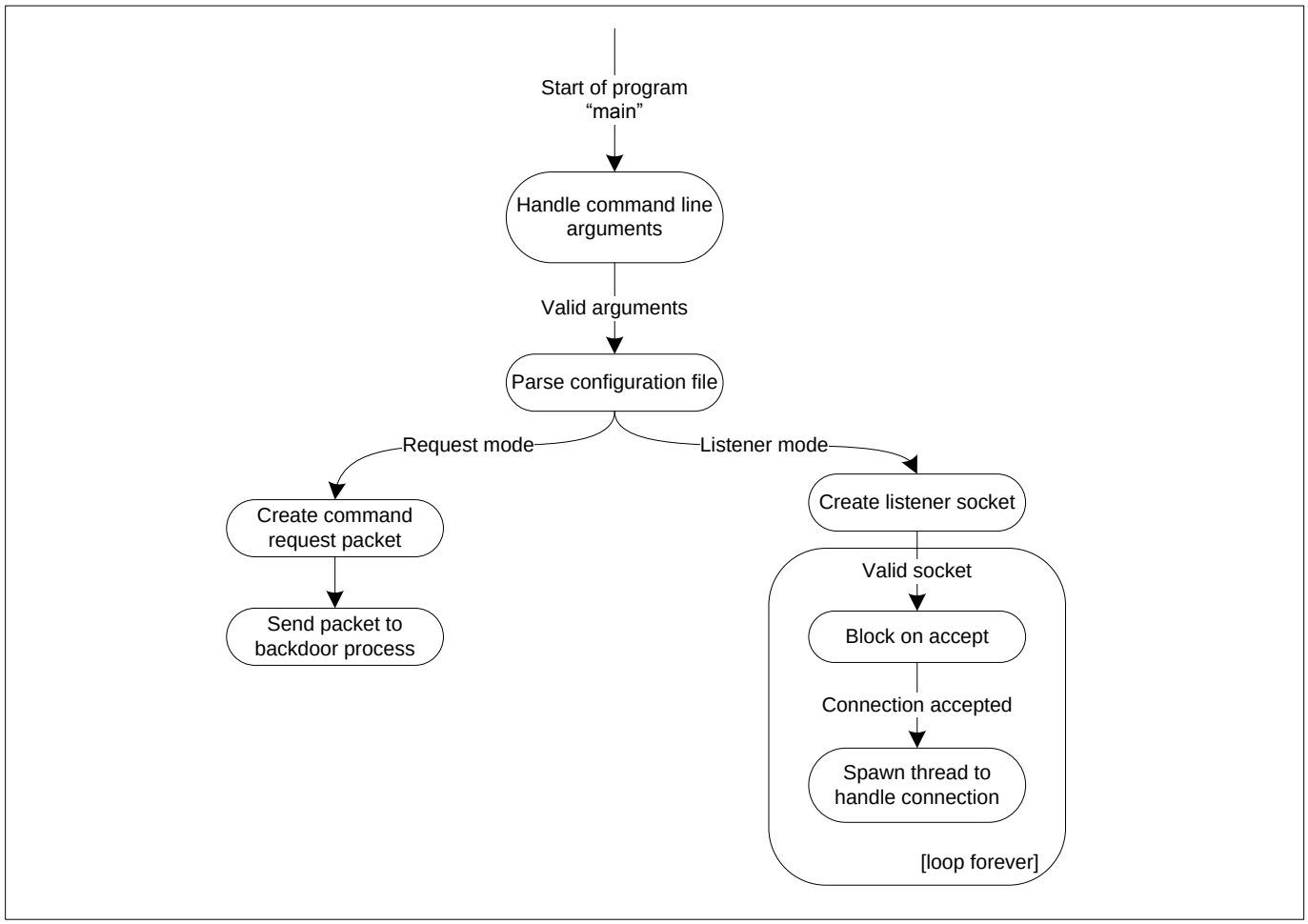


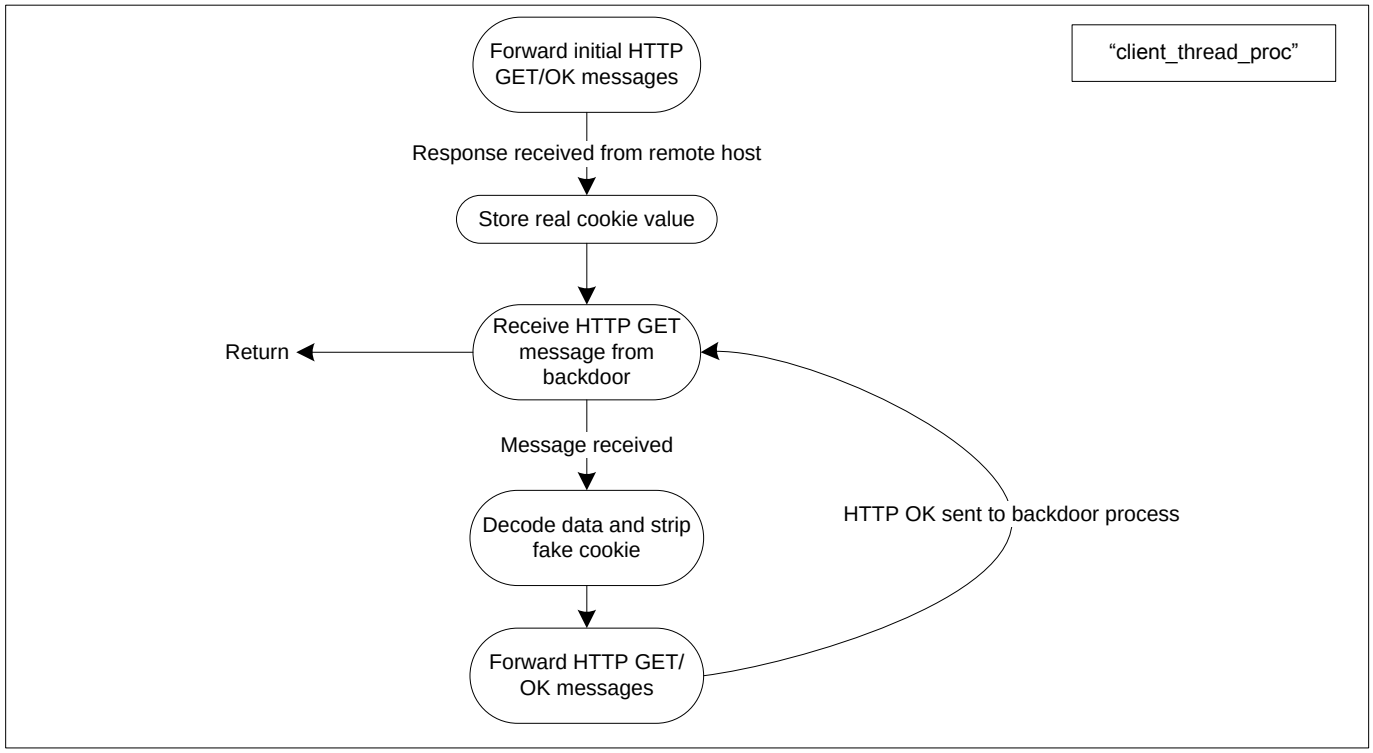



Timeout occurs
Check network usage
Below threshold
Threshold reached
Send initial HTTP GET request
Return
Response received
No more elements
Get element from queue
Got element
Sent response
Craft and send HTTP GET request


inotify event signaled
Return
Invalid descriptor
Open file descriptor
Valid descriptor
Push descriptor into response queue
ARP/ICMP packet received
Check target against spoofed host list
No match
Return
Found match
Respond to sender
HTTP traffic event
Check packet for substring
No match
Return
Found match
Spawn keylogging thread
thread
Get I/O port permission
Return
Got port
Read bytes for N seconds
Time up
Push keystroke buffer into response queue

Client Architecture





Pseudo-code Listings

Server Functions

"main"

```
handle command line arguments
parse configuration file
create pcap descriptor for command requests and associate with sched_event structure
(callback="command_request_cb")
create pcap descriptor for ARP requests and associate with sched_event structure (callback="arp_cb")
create pcap descriptor for ICMP requests and associate with sched_event structure
(callback="icmp_cb")
create epoll descriptor
add pcap descriptors and associated structures to epoll descriptor
invoke "scheduler"
```

"scheduler"

```
loop forever:
        block on "epoll_wait" with timeout of N seconds
        if timeout occurs: invoke "check_usage_cb"
        if error occurs: exit process
        if successful:
                for each epoll event:
                        invoke "callback" associated with epoll event data
                        if "callback" fails: invoke "cleanup" associated with epoll event data
```

"command_request_cb"

```
read N bytes into buffer from pcap descriptor
if amount read equals -1: return FAIL
decrypt buffer from ascii armor
if "check_timeout" fails: return SUCCESS
if "check password" fails: return SUCCESS
if opcode equals exfiltration request: invoke "handle_exfil"
if opcode equals keylog request: invoke "handle_keylog"
if opcode equals system call request: invoke "handle_system_call"
return SUCCESS
```

"arp_cb"

```
if packet type is not ARP request: return
for each spoofed host address:
        if ARP target equals spoofed host:
                create ARP response
                send ARP response
```

"icmp_cb"

```
if packet type is not ICMP ping request: return
if destination IP address is compromised host: return
create ICMP ping response
send ICMP ping response
```

“check_usage_cb”

```
if "check_throughput" fails: return
connect to client forwarder
send HTTP GET request for index (blocking)
read HTTP OK response (blocking)
for each element in response queue
        invoke "callback" associated with element
```

“file_event_cb”

```
close inotify descriptor associated with file
invoke "queue_file_for_exfil"
```

“send_file_cb”

```
loop until end of file:
        read N bytes into buffer from file
        encrypt buffer with ascii armor
        create HTTP GET request from keyword list
        inject buffer into HTTP get request
        write HTTP GET request to client
```

“send_keystrokes_cb”

```
encrypt keystroke buffer with ascii armor
loop until no more bytes in buffer
        create HTTP GET request from keyword list
        inject portion of keystroke buffer into HTTP GET request
        write HTTP GET request to client
```

“keylog_traffic_cb”

```
if packet does not contain substring: return
create thread with starting point at "keylog_proc"
```

“keylog_proc”

```
gain access to i/o port associated with keyboard
set timer to N seconds
loop until get timer fails:
        read key value from i/o port
        convert key value to ascii character
        append ascii character to key stroke buffer
associate key stroke buffer with queue_element structure (callback="send_keystrokes_cb")
```

“handle_exfil”

```
if requested file exists: invoke "queue_file_for_exfil"
if requested file does not exist: invoke "create_watch_for_file"
```

“handle_keylog”

```
create pcap descriptor for HTTP traffic and associate with sched_event structure
(callback="keylog_traffic_cb")
add pcap descriptor and associated structure to epoll descriptor
```

“handle_system_call”

```
invoke "system" with string received from command request
```

“create_watch_for_file”

```
create inotify desriptor and associate with sched_event structure (callback="file_event_cb")
add inotify descriptor and associated structure to epoll descriptor
```

“queue_file_for_exfil”

```
create file descriptor and associate with queue_element structure (callback="send_file_cb")
push structure into response queue
```

“check_timestamp”

```
if timestamp has impossible value: return FAIL
if timestamp has been seen already: return FAIL
add timestamp to list
return SUCCESS
```

“check_password”

```
if password is valid: return SUCCESS
return FAIL
```

“check_throughput”

```
retrieve current packet sent/received count
calculate packet sent/received rate
if rate is below threshhold: return FAIL
return SUCCESS
```

Client Functions

"main"

```
Handle command line arguments
Parse configuration file
If listener mode equals true:
        Create listening socket (port 80)
        Loop forever:
                Block on accept call
                Create thread with starting point at "client_thread_proc"
If request mode equals true:
        Build command request packet
        Write command request packet to backdoor process
Cleanup memory
Close descriptors
Return
```

"client_thread_proc"

```
Invoke "http_read" to receive initial HTTP GET from backdoor process
Invoke "http_parse_get_for_hostname" to retrieve remote hostname
Connect to remote host
Invoke "http_write" to send initial HTTP GET to remote host
Invoke "http_read" to receive initial HTTP OK from remote host
Store the cookie value in HTTP OK
Invoke "http_write" to send initial HTTP OK to backdoor process
Loop until "http_read" returns 0 or -1:
        Build character buffer from cookie fields
        Decrypt buffer
        Write buffer to stdout/file
        Replace cookie with initial cookie values
        Invoke "http_write" to send HTTP GET to remote host
        Invoke "http_read" to read HTTP OK from remote host
        Invoke "http_write" To send HTTP OK to backdoor process
Return
```

Conclusions

Defenses

- Deep Packet Inspection (DPI) is the single most important defense against application layer covert channels.
    - For example, by inspecting the “Cookie” field and matching it against expected values a system could alert analysts about potential information leaks.
    - While on the fly DPI may not be possible for high volume networks, outgoing data could be logged and analyzed on a nightly basis to detect possible information leaks as soon as possible.
    - Traffic analysis tools could be used to notify analysts of strange traffic patterns relating to specific network hosts.
- Tighten trust boundaries by removing network connectivity from security critical hosts or at the very least deploy IDS systems which perform DPI on traffic coming from these hosts.
- Be mindful of the types of traffic which are common on the network and which should never be seen.
    - For example, it may be highly unlikely that single UDP packets will be seen entering the network from unknown hosts during periods of high traffic.
- Attempt to restrict access to system tools such as insmod by removing them from the system entirely.
- Deploy security tools to verify kernel and system integrity as best as possible.

Areas of Improvement

- Remove any need for a user land process to handle command requests by implementing all Socket I/O in kernel land.
    - This would also have the side benefit of hiding the exfiltration socket presence from tools such as netstat.
- Web crawl the Google search responses to add another layer of misdirection.
- Match the User-Agent field in command response packets to the sender’s system.