

## **Network Stream Debugging with Mallory**

Jeremy Allen <[jeremy.allen@intrepidusgroup.com](mailto:jeremy.allen@intrepidusgroup.com)>

Rajendra Umadas <[raj.umadas@intrepidusgroup.com](mailto:raj.umadas@intrepidusgroup.com)>

## Abstract

Testing software for security bugs and flaws that do not use a standard, well known, protocol can be very challenging. On a time-boxed test where a client and a server, or a set of peers, are the targets of an assessment, a tester faces tough decisions. When the security tester's tools, such as debugging proxies, cannot easily Man in The Middle (MiTM) the application's traffic this eliminates valuable time the tester could spend attacking the application itself. Very often these applications are protected with some form of cryptography to provide point-to-point data protection, SSL being the industry standard for this task. The application that is being assessed may also be running on a mobile device, which may not provide a friendly environment for standard testing tools. Testers often spend the early portions of an assessment massaging application traffic into their favored tool chain for observing, manipulating, decrypting and generally Man in The Middling the traffic. Wrestling with SSL certificate chains, chaining socks proxies, spinning up netcat and socat forwarders and configuring HTTP proxies are all common activities in this phase of an assessment. What is a tester to do?

Contents

Abstract .....2

Contents .....3

Introduction .....4

Architecture.....6

    Protocols .....6

        HTTP.....6

        HTTPS.....6

        SSH .....6

        UDP .....7

        DNS.....7

        SSL.....7

    Plugins.....8

        HTTP Plugins.....8

        A Plugin Example: Session Hijacking .....8

        Other HTTP Plugins .....10

    Flexibility in Design.....11

Testing and Debugging .....13

    The GUI .....13

    Rules .....14

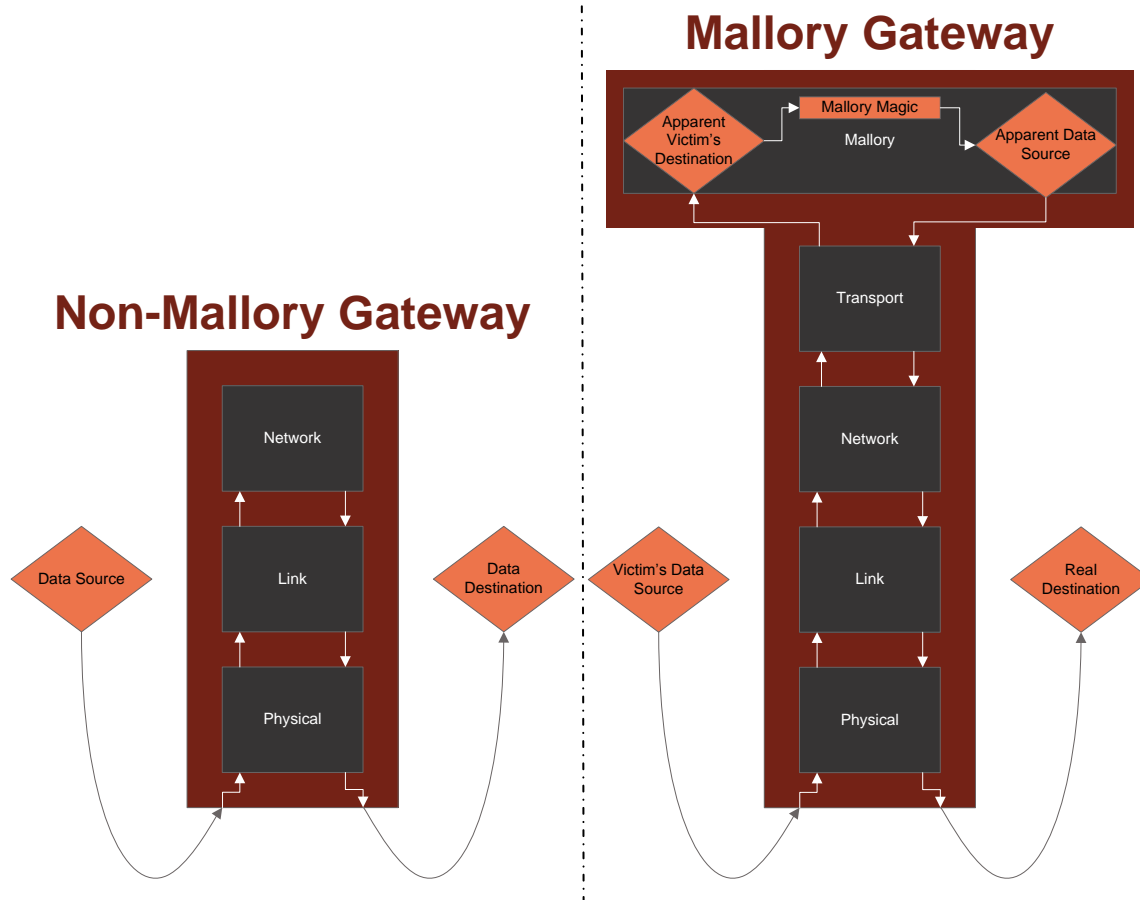
Bibliography .....18

## Introduction

Mallory is an idea, and now a tool, that came about due to frustration with many of the tools out there. There are many well known, and trusted, MiTM tools. These tools often focus on being an all in one solution: performing ARP poisoning to intercept traffic, routing the traffic, decoding the traffic and providing data modification facilities for traffic. The constraints placed on tools of this nature often make them less than ideal for an assessment focusing on one specific application or protocol. Mallory's design discards several aspects of many MiTM tools, and focuses on the transport and application layer aspects of MiTM. Mallory provides a straightforward tool for getting in between SSL traffic, performing MiTM on it and modifying it in transit. In fact, there is no code in Mallory for working with Link or Internet layers of the traffic. When Mallory began life, it was to be an internal tool. The goal was to design a tool, for application assessments, which could intercept, decrypt and record traffic quickly and easily. Mallory has since grown into a sizeable security tool that solves many problems security testers encounter when assessing applications. Additionally, Mallory also provides many traditional MiTM features. We consider Mallory a valuable addition to our arsenal.

We make several important assumptions about the design and usage of Mallory:

- An entire system must be a dedicated Mallory Gateway
- How the traffic reaches the Mallory system is not something Mallory really cares about
- Mallory will only work with and understand Transport Layer (OSI Model) protocols, primarily TCP and UDP
- Mallory (currently) runs on Linux and has access to Netfilter connection tracking
- Mallory will be implemented with a high-level language that lets us focus the problem at hand (Python).



**Figure 1: Non-Mallory to Mallory Gateway Comparison. (Red= Host, Grey= Independent layer, Orange= Data Consumer/Generator)**

These assumptions are not onerous in the age of cheap computing power and readily available virtualization technology. The assumptions outlined greatly influenced and shaped the architecture of Mallory.

Another important consideration that shaped the design of Mallory was mobile application testing, mobile applications often possess characteristics that make them quite difficult to MiTM. There are many reasons for the degree of difficulty: they might not have configuration options for proxies, applications might not be proxy aware and running proxies on a device can be challenging given time constraints. However, most mobile devices can connect to a WiFi access point, and do send application data over that link. Creating a proxy that could transparently proxy this type of traffic, and virtually any other traffic on WiFi or Ethernet link, was the original purpose of Mallory.

## Architecture

Mallory is a platform for implementing protocol awareness, which will be MiTMed and plugins for those protocols. Additionally, Mallory supports a GUI that can configure many aspects of Mallory's behavior and debug/modify TCP streams. Python is a natural choice because there is a vast ecosystem for Python and the language is hacker friendly. Every common protocol has an implementation in Python, most of which can easily be leveraged in Mallory with a little work. The design of Mallory aims to provide just enough structure for protocols, plugins for protocols, and tasks like debugging to get the job done without getting in the way or being overly cumbersome. Most protocols do not work directly with the Debugging GUI, SSL being the notable exception. Protocols are designed to work with plugins and provide a framework for plugin development that allows developers to automate and easily implement many common MiTM features. Mallory is, essentially, a transparent MiTM proxy that resembles other transparent proxies, such as Squid (Wiki, 2010) and many COTS proxies used by large commercial entities and government organizations.

## Protocols

Protocols are an important concept in Mallory, roughly analogous to common protocols, such as HTTP, SSL, SSH and other common application layer protocols. They represent one of the methods Mallory can work with intercepted traffic. Protocols focus on active traditional MiTM tasks that require full knowledge of how to decode and intercept a network stream to do something interesting from the MiTMs perspective. A tool that forwards traffic would not be very interesting from design and implementation standpoint. Mallory protocols decode traffic and encapsulate it into programmer friendly objects that are easy to manipulate. Once Mallory decodes traffic, it is accessible to the plugin system, while providing a plugin author with a more abstract view of the data. There are several protocols already implemented in Mallory. The currently featured protocols are HTTP, HTTPS, SSL, SSH and DNS. With these core Internet protocols properly implemented it is possible to decode and understand the traffic of many applications. When none of the implemented protocols are appropriate or Mallory encounters an unknown protocol the lower level TCP and UDP protocol implementations are recruited to handle the traffic.

### HTTP

The HTTP protocol is a simple protocol in that it is stateless and there is a single request and a response. Mallory captures both the request and the response and decodes them into objects. These objects provide access to the key parts of an HTTP request/response, such as the headers and the body. The protocol also decodes various encoding types, such as chunked encoding. HTTP is everywhere and was obvious choice for an initial protocol due to its ease of implementation and the huge base of applications that support and use HTTP for at least some amount of functionality.

### HTTPS

HTTPS is HTTP with SSL mixed in. Many applications use HTTP with SSL and this was also an obvious protocol to support. It behaves identically to HTTP with regard to HTTP protocol decoding.

### SSH

SSH is a more interesting protocol to implement, SSH is primarily an interactive protocol that requires long running TCP connections. The approach taken with this protocol was to use the

Paramiko (Pointer) library. For each intercepted SSH connection an SSH daemon is spawned that attempts to MiTM the SSH connection. Upon successful MiTMing, the victim's username and password (in cases where password authentication is used) are logged. Mallory provides a simple TCP based server that can then be accessed via telnet or Netcat. This server lists all known MiTMed SSH sessions and lets the user open a shell on that SSH transport. Because the existing transport is reused a new channel with a PTY is opened without creating a new connection to the server. In log files it will only look as though the user has authenticated once. It is also possible to capture and inject arbitrary content into the SSH session of the victim.

Another feature of interest is that any port forwarded connections can be intercepted via custom methods to perform tasks, such as MiTM of an SSL connection forwarded via SSH.

```

mallory@mallory:~/Dev/Mallory/src$ telnet 127.0.0.1 20756
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
[0] SSHProtocol connected to ('65.254.211.221', 22)
Which SSH MITM session would you like a shell on? 0
Last login: Mon Jul  5 15:26:39 2010 from
sshtest@crackedrearview:~$ █

```

**Figure 2: SSH MiTM in action. Mallory provides the server on port 20756 by default.**

## UDP

Mallory has full support for UDP. Transparently proxying UDP based protocols is an interesting task for a proxy. UDP is a connectionless, stateless protocol. Any protocol that has or maintains state on UDP does it with its own state maintenance mechanism. Many of the most famous examples of UDP based protocols are short-lived request and response oriented protocols, such as DNS. Some protocols will implement a state mechanism, which usually involves frequent “keep alive” datagrams to let the client/server know the connection is still active. If the client fails to send a “keep alive” datagram the server will assume the connection has died and close the session. Mallory assumes a reasonably long timeout for such “keep alive” datagrams, which enables Mallory to reasonably proxy almost all UDP based protocols. Currently, the only implemented UDP protocol for is DNS.

## DNS

DNS support in Mallory is limited to A record manipulation. Currently, Mallory supports a mapping table that will return the IP address specified for a specific DNS A record request. The table supports wildcard style matching of the queried hostname. This functionality is implemented using the excellent DNS toolkit for Python: dnspython (dnspython). Due to the extensive dnspython library, it is possible to extend the DNS functionality rapidly to include other query types, depending on the needs of the tester. However, this will currently require further development of the DNS protocol. The common use case for the DNS protocol during testing is to return a local IP address for domain names that have been statically compiled into, or configured into, a binary or application. Once the traffic is headed to a local address it can be recorded or the tester may attempt to emulate the server locally for expedient testing of the application.

## SSL

SSL support in Mallory is based on the python SSL package, which is in turn implemented using OpenSSL. Mallory deploys with a Certificate Authority (CA), which is used for all SSL man in the middle operations. Using previously outlined techniques, a developer can easily use the SSL protocol as a “mixin” to create custom SSL based protocols. Alternatively, the SSL protocol can be used naked, with the base TCP protocol implementation in Mallory, to provide a debugging session for SSL using the Mallory TCP Debugging GUI.

## Plugins

The plugin framework provides a standard interface to alter the behavior of protocols at key points in each protocol. One of the challenges of generalizing plugin architecture is that each protocol functions differently and has different places where a plugin action makes sense. Any point in a network flow where a plugin could hook may be a good place to read and modify data, only read data or only modify data. Each protocol is different. To address this each protocol generates different events, which the plugin can then respond to depending on the event. To support plugins each protocol has one protocol daemon. The protocol daemon runs in its own thread and is responsible for managing the plugins for that protocol. When a plugin must be instantiated the daemon will instantiate it. The daemon is also responsible for brokering communication between protocols and plugins. A protocol never directly communicates with a plugin. This provides the adequate abstraction for plugin architecture, which lets the plugin developer focus on their plugin.

### HTTP Plugins

One of the most common protocols used by applications, even thick clients and other client to server applications is HTTP. It is everywhere and used in all manner of environments. It was the first protocol implemented in Mallory. The protocol, primarily, decodes HTTP requests and responses and provides easy access to the HTTP headers and body. The HTTP plugin has the opportunity to collect and modify data from requests and responses. Plugins that collect data can often provide an interesting outlet for programmatic analysis of the data as it flows. Applications often send sensitive data over HTTP and HTTPS. In particular, session identifiers, Personally Identifying Information (PII), and username’s and passwords are of interest. A plugin that collects this data for analysis and observation can be tremendously valuable over the course of an application assessment. Data modification is another valuable activity in HTTP. In cases where some logic that is more complex than a regular expression, plugins provide a perfect solution for a security testers needs. For instance, in many assessments the tester may need to replace certain values in each request. A plugin easily automates this task. This architecture also lets a plugin author implement complicated plugins. A plugin could easily implement a feature SSL stripping tool, such as Moxie Marlinspike’s sslstrip (Moxie). The advantage of a Mallory plugin versus a stand-alone MiTM tool is easy to see: a plugin author can focus on their idea and code and not worry with the lower level details of protocol decoding and network handling.

### A Plugin Example: Session Hijacking

The session hijacking plugin, currently, collects certain header values (the Host and Cookie headers) values and stores them in memory. Plugins can store data in any manner they choose: memory, files, databases, other network services, etc. Plugins are, essentially, python code with no limitations beyond those presented by the physical hardware. Using another, optional, feature of the plugin architecture, it also has a listening server that can pass all of the collected headers to a client. Typically, to hijack a session all that is required is the Host header and the Cookie



header. To facilitate session hijacking the plugin collects these two header values for each request that flows through the proxy. For Mallory and security testing needs, it was straightforward to implement a browser extension as our initial client agent. The browser extension queries a service running in the Mallory process. The service returns data formatted in JSON (Wikipedia).

```

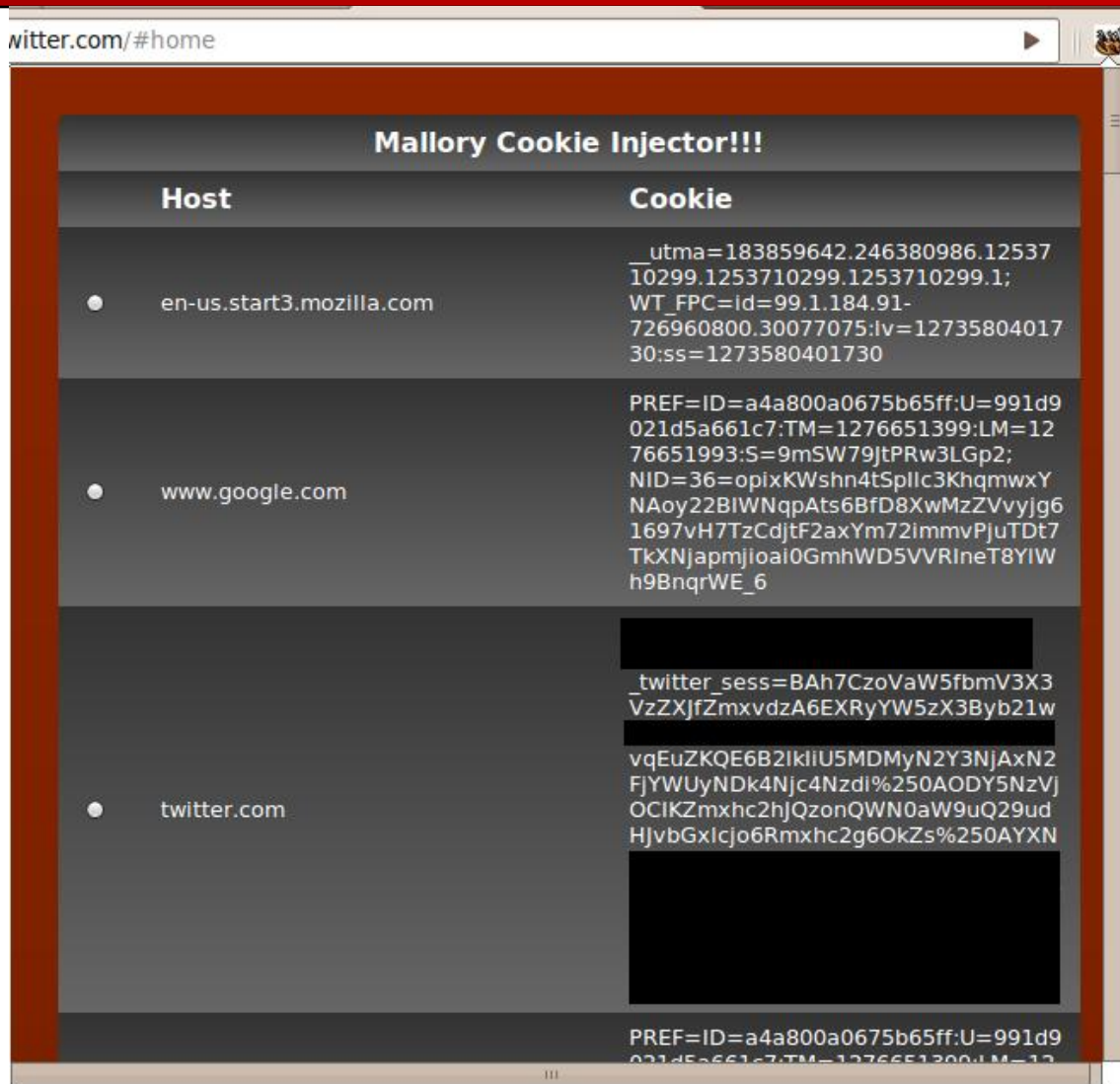
mallory@mallory: ~ 80x21
mallory@mallory:~$ nc 127.0.0.1 20666
GET / HTTP/1.1
HTTP/1.0 200 OK
Content-Type: text/html

[[{"Host: en-us.start3.mozilla.com", "Cookie: __utma=183859642.246380986.1253710299.1253710299.1253710299.1; WT_FPC=id=99.1.184.91-726960800.30077075;lv=1273580401730:ss=1273580401730"}, {"Host: www.google.com", "Cookie: PREF=ID=a4a800a0675b65ff:U=991d9021d5a661c7:TM=1276651399:LM=1276651993:S=9mSW79JtPRw3LGp2; NID=36=opixKWshn4tSpIlc3KhqmwYNAoy22BIWNqpAts6BfD8XwMzZVvyjg61697vH7TzCdjtF2axYm72immvPjuTDt7TkXNjapmjioai0GmhWD5VVRIneT8YlWh9BnqrWE_6"}, {"Host: twitter.com", "Cookie: k=99.1.184.91.1278177159274569; guest_id=127817715927717192; _twitter_sess=BAh7CzoVaW5fbmV3X3VzZXJfZmxvdzA6EXRyYW5zX3Byb21wdDA6D2NyZWf0%250AZWRfYXRskWhvqEuZKQE6B2lkIiU5MDMyN2Y3NjAxN2FjYWUyNDk4Njc4Nzdi%250AODY5NzVjOCIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g60kZs%250AYXNoSGFzaHsABjoKQHVzZWR7ADoMY3NyZl9pZCilYTkyOWY5OGE0YTY0MDdj%250AMDc0ZW20TU5YWYyZmZmZmI%253D--837b40571f5a488806b1e026fa9902e8c4e1fa32"}, {"Host: maps.google.com", "Cookie: PREF=ID=a4a800a0675b65ff:U=991d9021d5a661c7:TM=1276651399:LM=1276651993:S=9mSW79JtPRw3LGp2; NID=36=opixKWshn4tSpIlc3KhqmwYNAoy22BIWNqpAts6BfD8XwMzZVvyjg61697vH7TzCdjtF2axYm72immvPjuTDt7TkXNjapmjioai0GmhWD5VVRIneT8YlWh9BnqrWE_6"}, {"Host: twitter.com", "Cookie: k=99.1.184.91.1278177159274569; guest_id=127817715927717192; _twitter_sess=BAh7DjoVaW5fbmV3X

```

**Figure 3: The Session Hijacking plugin implements a simple TCP server that always returns one HTTP response. The response returned is JSON data that can be returned to any client that wishes to acquire the host and cookie value pairs. For security reasons only clients from localhost are allowed to connect.**

Currently, an extension is implemented for Chrome. This extension queries the service by sending it a standard HTTP GET request. It then formats this data in the browser to perform the session hijacking. The chrome extension has been configured to work outside of the same-origin-policy of the Chrome browser and thus is able to set a cookie for any domain.



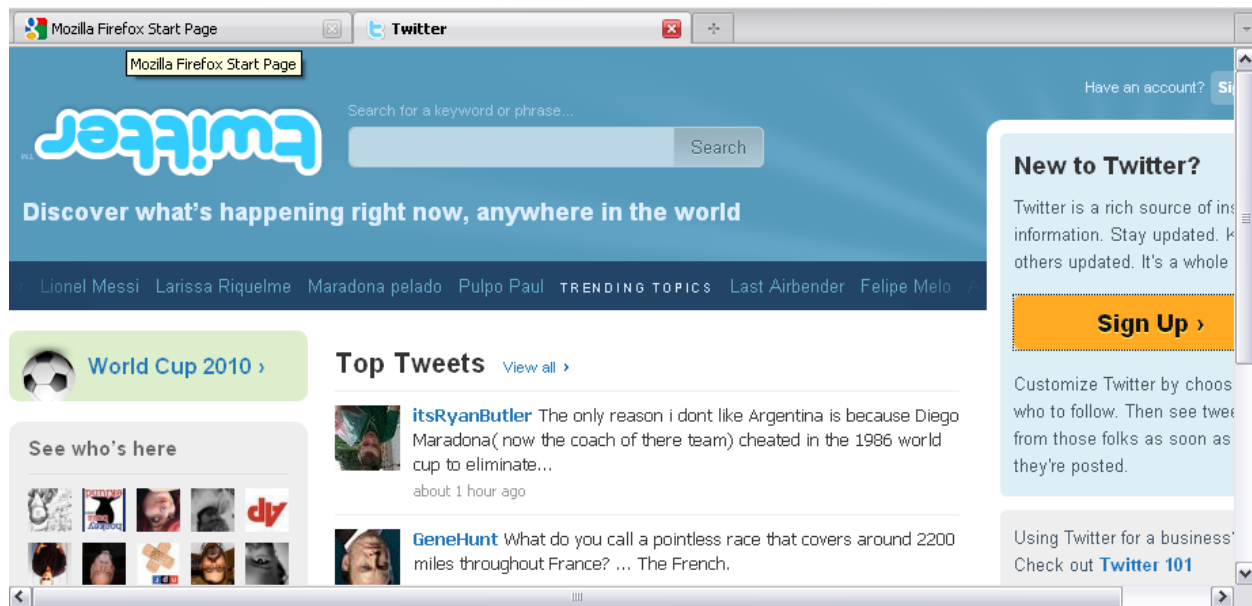
**Figure 4: The chrome extension in action. This lists all of the cookies Mallory has captured and lets the user select the cookie they wish to apply.**

The flow for using this extension is to review the list of captured cookies and identify the appropriate cookie/host pair. The user then clicks the radio button and then the Apply button (not shown in the figure). The user must have the targeted domain open in a tab. When the apply button is clicked the cookies for that host are set from the extension and the page is reloaded. Session hijacking in three easy clicks. This is very valuable for testing certain behaviors of a session management system. For assessments that focus on web applications consumed from mobile device users, this feature makes it easy for the tester to load the application in a standard browser and use a typical HTTP proxy tool chain for testing the web application. Additionally, demonstrating this extension and plugin is an effective reminder of the risk of XSS and poor session management mechanisms.

### Other HTTP Plugins

There are many other HTTP plugins. Another key plugin example, mostly done for the entertainment of Mallory's authors, was the "upside down Internet" plugin. Implemented in a very small amount of code, this plugin flips images upside down using the Python Imaging Library (PIL). We developed this plugin to highlight the Python code ecosystem, ease of plugin

implementation, and to make web application penetration testing more entertaining. The figure below shows the effect of this plugin running.



**Figure 5: Upside down Internet. There are many creative data modification uses for the plugin system.**

## Flexibility in Design

Another key consideration when designing any software system is flexibility. Striking a good balance between design, refactoring and simply getting things done is a continual struggle for a programmer. There must always be a balance, and the Mallory architecture is kept relatively clean to create a platform that allows protocols and plugins to be implemented with an intuitive API for the protocol. Protocols require more work to implement as they span a huge variety of tasks and purposes. Mallory provides a protocol implementation with connected and ready to go source and destination socket objects. What the protocol does with those sockets is up to the protocol implementer. Many Python libraries that implement protocols expect, or can at least use, sockets that are ready to go.

One of the key aspects of the design is that each protocol has the opportunity to respond to certain critical events in the life of a TCP stream. These events are: server socket creation (SSCREATE), client socket after accept (CSACCEPT), client socket after server socket creation (CSAFTERSS), start data forwarding for client to server (STARTC2S) and start data forwarding for server to client (STARTS2C). The first three events deal with socket initialization and give a protocol implementer a chance to modify the socket itself, such as wrapping it in a custom cryptography protocol that is transparent to the data forwarding methods. The latter two events each run in their own thread and forward the data in the indicated direction. It is important to note that there are two threads, while there is only one protocol instance for those two threads. Because of this, thread safety considerations are of critical importance when reading or writing to member objects of the protocol instance.

One example of this architectural flexibility in action is the HTTPS protocol. It is a mixin of the HTTP and SSL protocols, mixins in python are usually some form of multiple Inheritance. The

figure below illustrates the simplicity of wrapping any protocol in SSL. That is the approach taken in the Mallory HTTPS protocol.

```

1 from base import TcpProtocol
2 import sslproto
3 import http
4 import malloryevt
5
6 class HTTPS(http.HTTP, sslproto.SSLProtocol):
7     """
8     """
9     def __init__(self, trafficdb, source, destination):
10         http.HTTP.__init__(self, trafficdb, source, destination)
11         sslproto.SSLProtocol.__init__(self, trafficdb, source, destination)
12         self.serverPort = 443
13         self.name = "HTTPS"
14         self.log.debug("HTTPS: Initializing")
15         self.supports = {malloryevt.STARTS2C:True, malloryevt.STARTC2S:True,
16                         malloryevt.CSAFTERS2S:True, malloryevt.SSCREATE:True}

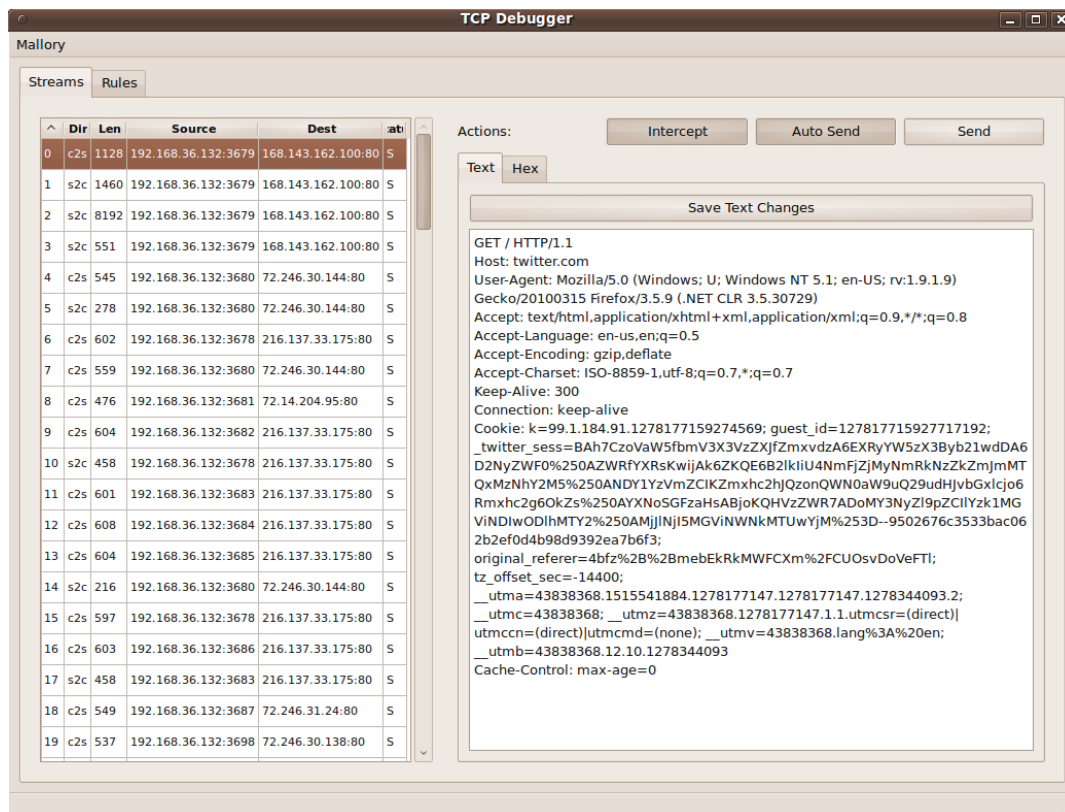
```

**Figure 6: The HTTPS plugin. This plugin is a mixin and does not actually implement any methods. All of the required methods are inherited from the HTTP and SSL protocol implementations. The inheritance is performed on the class declaration, seen on line 6**

This gives an author great flexibility in implementing a protocol. For example, if a Mallory developer has implemented a protocol, but an application has wrapped that protocol in SSL, using a program such as Stunnel (Hatch) or linking against and using OpenSSL (OpenSSL) internally, the developer can mix in the SSL protocol to created “secured” versions of the protocol.

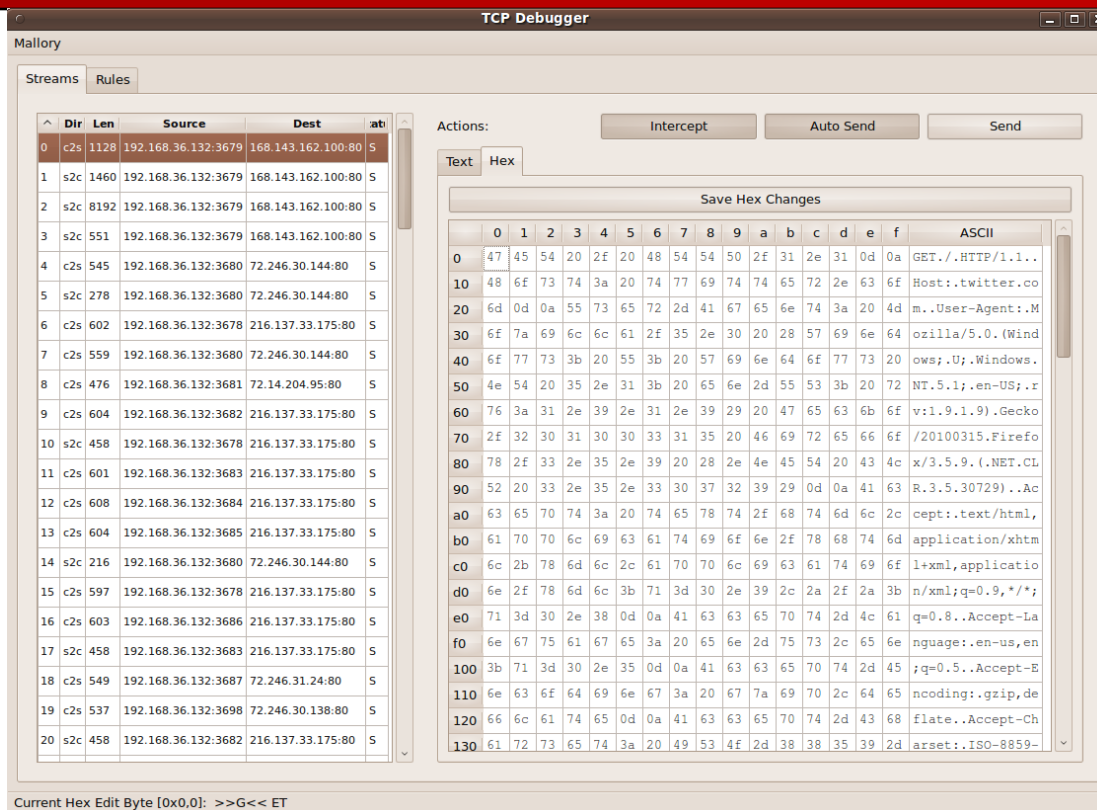
When a protocol is available or known and it is common, the job of testing and debugging an application using it is much less complicated. When the application is employing an unknown protocol, it is often significantly more difficult to assess the application in a time-boxed assessment. Not only must the tester reverse engineer the protocol, but they must also assess the application and what it does. Just viewing the traffic can become a significant challenge. One of the frustrations of the author's of Mallory was that most tools required, at a minimum, socks or HTTP proxy support, or the ability to control the IP address of the server a client tried to connect to. In many cases, these constraints are impractical or can be time consuming, if the tester must reverse engineer and patch a binary just to change a destination address. One of Mallory's key features is the ability to easily debug, and modify, data from any TCP stream. Mallory has a rule system to refine the data sent to the Mallory debugging client. The rule system also supports a data modification system known as the "Muck Pipe".

A GUI can dramatically simplify some tasks. Mallory supports a simple XML RPC interface for all debugging clients. The debugger (Note: The debugger is called such even though it supports many features that are not very debugging-like) receives all of the traffic it is configured to receive and gives the user a chance to view, and modify it. There are two views available for the data: text or hex. In many protocols a text view of the data is inappropriate (the protocol supports null bytes). However, the text view has its uses. The figure below illustrates the standard GUI interface in Text Edit mode.



**Figure 7: Text view mode of an HTTP GET Request**

And now, the same request viewed in Hex Edit mode.



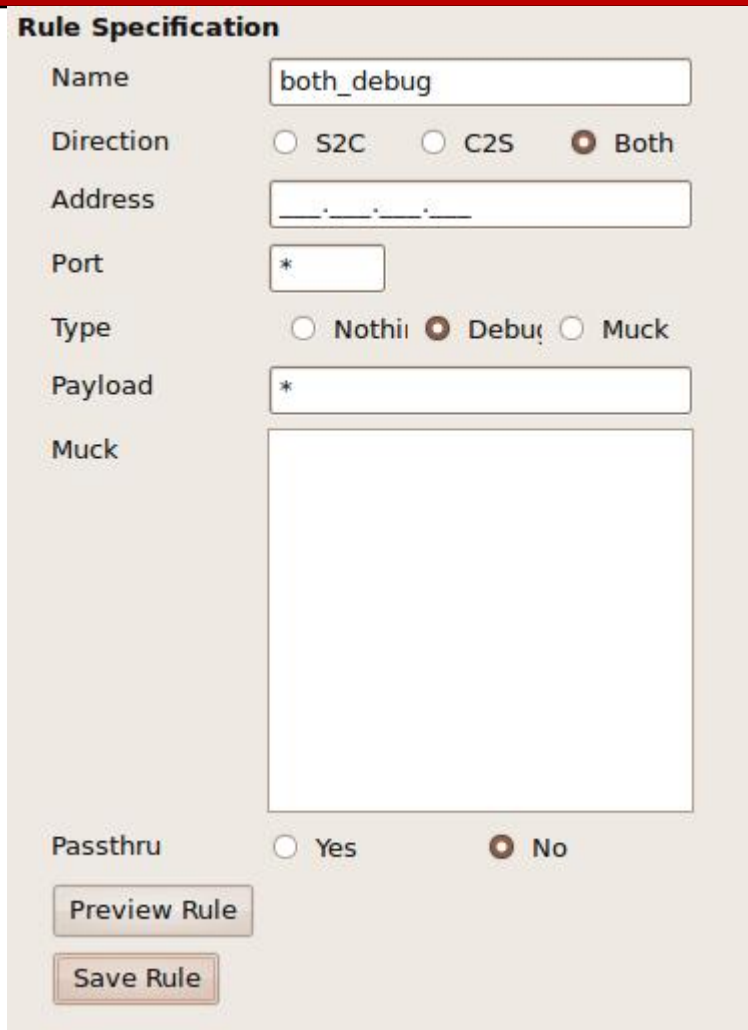
**Figure 8: An HTTP Get Request in Hex Edit Mode**

The user interface controls if data is intercepted or automatically sent. If the TCP debugger does not automatically send it, the user must send each set of stream bytes by clicking the send button, or pressing the S key. This provides one of the essential features of Mallory and provides a fast way to both view and edit the data. Tools like Wireshark (Wireshark) already provide an excellent view only functionality for a data passing through a MiTM gateway, but it does not provide a method to pause and modify the data that flows through it. This is the TCP debugger's job.

## Rules

Early in the design of GUI, it became apparent that controlling what streams Mallory sends to the debugger would be important. There were two options considered: IPtables rules or an internal rule system for Mallory. The internal system is the current design. With IPTables the assumption is that the user would pick and choose specific IPs and ports and that is all that would get sent to Mallory. One of the key factors for the internal rule system is that it is significantly easier to use for quick rule changes and several other planned features have been incorporated into the rule system (including the muck pipe). Additionally, Mallory can be used for an entire interface (logging data and running data modification rules), while only sending a select set of traffic to the debugger. A rule is a set of matching conditions for data that has no configured protocol (using the base TCP protocol handler for unknown traffic). Each rule has an action or rule type. Mallory executes the action when the conditions for the rule match. Rules are stored in an ordered list and each rule has the option of terminating the processing or the rule list or passing through to allow lower priority rules to match. The pass through option for rules lets multiple different rules types such as a Muck (data modification rule) rule and a Debug rule (send to the debugger) to match the same data. The screen capture below shows the various options that rules can match on.





The image shows a 'Rule Specification' GUI with the following fields and options:

- Name:** both\_debug
- Direction:** Radio buttons for S2C, C2S, and Both (Both is selected).
- Address:** A text field with a placeholder pattern: \_\_\_\_.\*\_\_\_\_.\*\_\_\_\_.\*
- Port:** A text field containing an asterisk (\*).
- Type:** Radio buttons for Nothi, Debug (selected), and Muck.
- Payload:** A text field containing an asterisk (\*).
- Muck:** A large empty text area.
- Passthru:** Radio buttons for Yes and No (No is selected).
- Buttons:** 'Preview Rule' and 'Save Rule'.

**Figure 9: Rule specification GUI.**

The rule creator provides a friendly interface to edit, apply, delete, and otherwise configure rules in Mallory. All rules have a number of configurable options. Firstly, a set of matching conditions are available: direction, address, port and payload contents. The rule must also have a type, and if it is a Muck rule it must have a MuckPipe transformation. MuckPipe transformations are sed-like regular expressions that can transform the data using the Python regular expression system. Binary transformation is possible. The ordering of rules is important, if a rule is not pass through and it is a match for the incoming data the rule system stops processing rules. Note: rules can be saved in ruleconfig.py or edited in the GUI, ruleconfig.py is for persistent/permanent rules. Below is an example rule that sends all traffic to the debugger.

```
{
  "name": "default",
  "action": rule.Debug()
}
```

**Figure 10: Default debug rule that sends all traffic to the debugger. Rules are not pass through by default.**

For any rule match item that is not included in the rule configuration, it is assumed to be a wildcard configuration and will match any criteria for that incoming stream. If the tester only wanted data that was sent from the client to a server, the following modification could be made:

```
{
  "name": "default",
  "action": rule.Debug(),
  "direction": "c2s"
}
```

**Figure 11: Debug rule specificity increased to match only on the client to server direction**

Rules can also match on IP address, TCP destination port and payload contents (simple string matching).

## Muck Pipe Rules

The MuckPipe is similar to features in other MiTM proxies and was designed to be very quick and simple to create in the tradition of Unix command line stream editor sed. The MuckPipe is a rule action that executes when a rule is matched and a properly formed Muck action is assigned to the rule. A MuckPipe is a series of sed-like stream edits that the rule system executes on a stream. Below is an example of a MuckPipe Rule that will remap the upper case "A" key to the upper case "B" key for the VNC protocol:

```
{
  "name": "c2s_vnc",
  "direction": "c2s",
  "port": "5900",
  "action": rule.Muck([ "\x04\x01\x00\x00\x00\x00\x00\x41/\x04\x01\x00\x00\x00\x00\x00\x42/g" ]),
  "passthru": "true"
}
```

**Figure 12: MuckPipe transformation to remap keys in VNC. This would be executed in the client to server direction to remap the key the client sent. Note that this is a rule specification in the persistent ruleconfig.py, which is where long term / permanent rules can be saved.**

The syntax is not easy to spot with all of the backslashes milling about. Below is a more simple MuckPipe rule for HTTP clients that strips gzip and deflate strings from a client, which tells the server the client can't handle compressed content. When the content comes back from the HTTP server it will, hopefully, be uncompressed for easy modification.



```
{  
  "name": "http_muck_mangle_c2s",  
  "port": 80,  
  "action": rule.Muck([ "gzip,deflate/ /1", "deflate/ /1", "gzip/ /1" ]),  
  "direction": "c2s"  
}
```

**Figure 13:** Note the syntax is “regular expression match”, then forward slash, then “replacement”, then forward slash, then “number of times to replace”.

There are two key points to note about a MuckPipe rule. It can apply multiple regular expressions (for those familiar with Python you will notice the second rule is actually three strings in a list, which applies three separate muck pipe transformations). The second point is that each muck pipe rule has a count of how many times it will match, and replace, data. Lower case g means global and a number means exactly this many replacements. The second rule, http\_muck\_mangle\_c2s only runs once to avoid modifying anything but header content if it matches. (If the header does not have this and the HTTP request body did, that content could get mangled). The muck pipe provides a programmatic way to make edits. This allows the tester great flexibility in replacing session identifiers, modifying values and performing other repetitive editing tasks on the fly.

**Bibliography**

*DNS Python*. (n.d.). Retrieved July 5, 2010, from DNS Python: <http://www.dnspython.org/>

Hatch, B. (n.d.). *Stunnel Landing Page*. Retrieved July 1, 2010, from Stunnel: <http://www.stunnel.org/>

Moxie. (n.d.). *SSLSTRIP*. Retrieved July 5, 2010, from SSLSTRIP:  
<http://www.thoughtcrime.org/software/sslstrip/>

OpenSSL. (n.d.). *OpenSSL Landing Page*. Retrieved July 5, 2010, from OpenSSL Project Page:  
<http://openssl.org/>

Pointer, R. (n.d.). *Python Paramiko*. Retrieved from Python Paramiko: <http://www.lag.net/paramiko/>

Wiki, S. (2010, February 19). *Interceptipn Proxy*. Retrieved July 5, 2010, from squid-cache wiki:  
<http://wiki.squid-cache.org/SquidFaq/InterceptionProxy>

Wikipedia. (n.d.). *JSON*. Retrieved July 5, 2010, from JSON: <http://en.wikipedia.org/wiki/JSON>

Wireshark. (n.d.). *Wireshark Home Page*. Retrieved July 1, 2010, from Wireshark:  
<http://www.wireshark.org/>