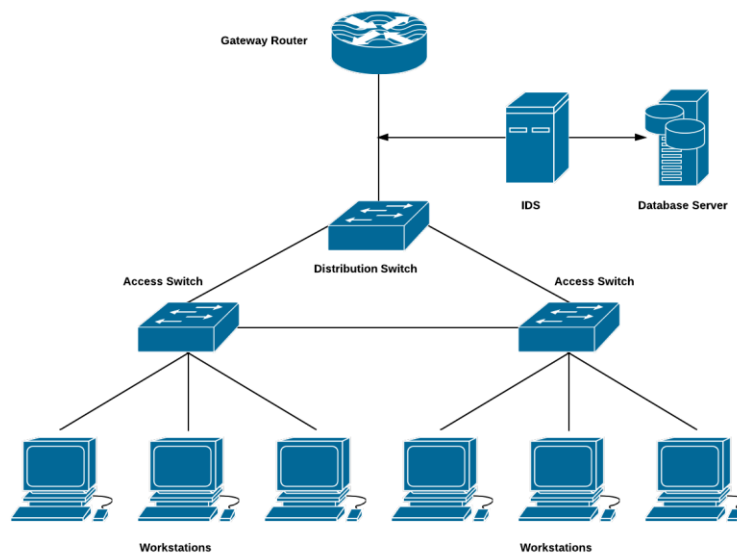


Table of Contents

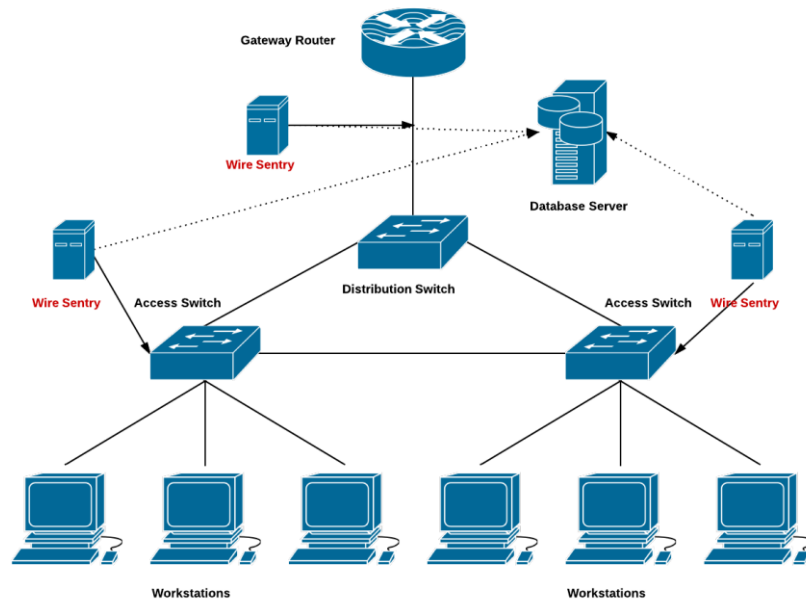
Introduction.....	3
Packet Capture Interfaces.....	5
Libpcap.....	5
SharpPCap.....	5
Logical Structure.....	6
File Layout.....	6
Startup Sequence.....	6
Command-Line Arguments.....	7
Packet Processing.....	9
Packet Capture.....	9
Packet Processor.....	11
Reverse DNS Lookup.....	12
Dynamic Modules.....	14
Scanners.....	14
Handlers.....	15
Module Manager.....	17
Logging.....	20
Attack Detection.....	22
ARP Spoof Attack.....	22
Port Scan Attack.....	24
Embedded Device.....	26
Resources.....	31

Introduction

Wire Sentry is an application framework and set of software tools for analyzing network traffic patterns to identify and log malicious network activity. The software actively captures network traffic and compares this traffic against various attack signatures in an attempt to identify illegitimate network activity. The main goal of Wire Sentry is to provide a lightweight, distributed, intrusion detection framework capable of scanning network activity from multiple vantage points on the local area network. While traditional intrusion detection systems have a broad view of network traffic, attacks performed at local switch access ports can generally go unmonitored without the extensive use of sensors. Such attacks include ARP spoofing, port scanning, and ICMP based reconnaissance attacks. Because Wire Sentry was designed to provide a lightweight and distributed IDS solution it can be loaded onto cheaper, embedded hardware devices and can monitor the network at local access switches. This implementation can detect attacks that may be missed by a traditional enterprise intrusion detection system.



(Traditional Network with IDS Setup)



(Wire Sentry Network Setup)

The software is built with a modular architecture in which attack pattern scanners can be added into the application via dynamically loaded software modules. The framework is distributed along with a SDK (software development kit) for creating add-on modules in a hope that a large library of attack patterns can be created. In addition to extensibility for attack scanners, the SDK also provides expandability through dynamically loaded modules called handlers. Handlers are software modules that define custom actions to execute when an attack is detected.

The framework and utilities are written in the C# language and have been compiled to run using the Mono Framework¹. The Mono Framework is a cross platform port of the popular Microsoft® .NET Framework. It allows managed code to be compiled and run on a variety of different operating systems and architectures. The Wire Sentry software utilizes the *libpcap*² API (application programming interface) and drivers to capture network traffic at the hardware level.

Packet Capture Interfaces

Libpcap is a popular system-independent interface that facilitates user-level packet capture. This interface provides a portable framework for network monitoring and packet capture. Wire Sentry utilizes the *libpcap* interface to analyze network traffic and perform packet captures on the network.

Wire Sentry is written using managed code, or code which runs exclusively under the management of a virtual machine. Using managed code has several important implications. Code written using a managed code environment can be easily ported between different CPU architectures and operating systems if an implementation of the appropriate virtual machine exists for that platform. For example, code written using the Mono Framework can be executed using the Mono Runtime on Microsoft® Windows, Linux, and Mac OS X regardless of the underlying CPU architecture. The disadvantage to using such an approach is that native operating system APIs and functions cannot be executed directly.

Because Wire Sentry is written using managed code it cannot access native libraries directly. Due to this software based limitation, the *SharpPCap*³ libraries are used to access the *libpcap* interface. *SharpPCap* is a Mono/.NET Framework wrapper for the *libpcap* interface which exposes the native functions to managed code for use.

Logical Structure

The Wire Sentry functionality itself is logically segregated into three software libraries. The first and largest of these libraries is the Wire Sentry daemon itself *WireSentry.exe*. This executable contains all of the main functionality of the Wire Sentry tool suite. It is responsible for starting the application, parsing command line options, dynamically loading scanner and handler modules, loading and configuring the *libpcap* interface, capturing network traffic, and scheduling scans of network traffic.

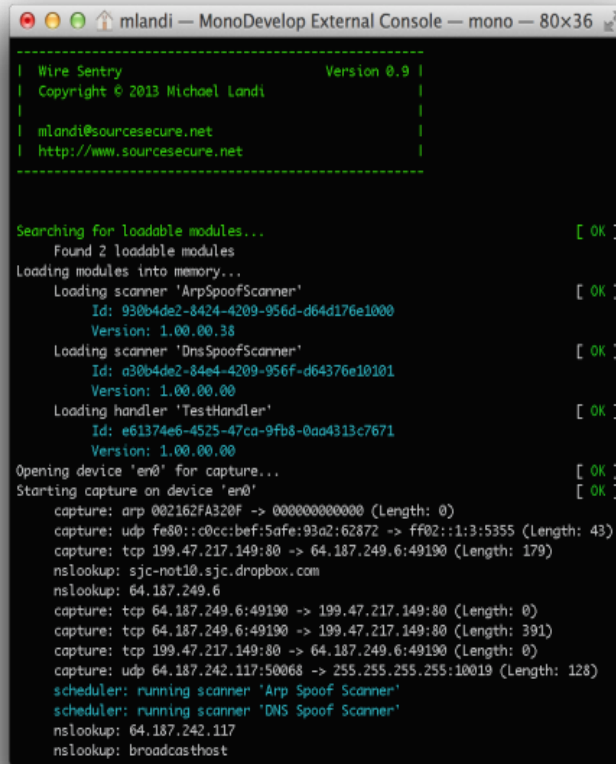
WireSentry.SDK.dll is a software library that contains an SDK (software development kit) for extending the functionality of the Wire Sentry application. It contains all of the necessary classes to build and deploy custom software modules that can be dynamically loaded into Wire Sentry to add additional functionality. Custom network scanners and handler modules can be created by utilizing the classes and interfaces provided by this library. All add-on modules must reference this library in order to be loaded dynamically by Wire Sentry.

WireSentry.Common.Scanners.dll is a software library that contains a standardized set of scanner modules for common network attacks. It contains several scanners to detect attacks performed in the local area network. The library must be located in the same directory as *WireSentry.exe* to be loaded into memory dynamically.

Startup of Wire Sentry occurs when the main executable *WireSentry.exe* is executed using the Mono Runtime. The startup sequence occurs as follows:

- Parse command-line arguments passed into the application.
- Establish a database connection for logging (optional).
- Locate loadable modules in the current directory.
- Load modules and schedule execution of scanner modules.
- Initialize the *libpcap* interface and begin packet capture.

- Wait for interrupt signal for application termination.



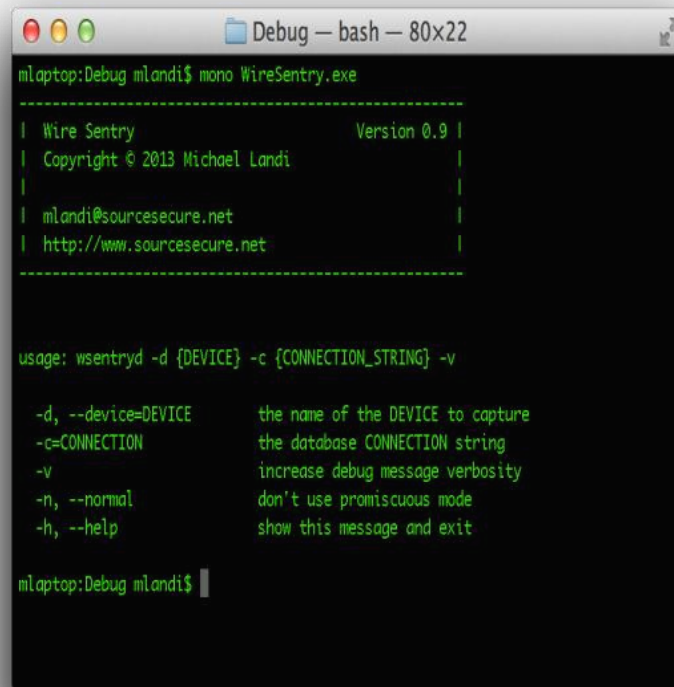
```
| Wire Sentry                               Version 0.9 |
| Copyright © 2013 Michael Landi           |
| mlandi@sourcesecure.net                 |
| http://www.sourcesecure.net             |
|-----|

Searching for loadable modules...           [ OK ]
Found 2 loadable modules
Loading modules into memory...
Loading scanner 'ArpSpoofScanner'           [ OK ]
  Id: 930b4de2-8424-4209-956d-d64d176e1000
  Version: 1.00.00.38
Loading scanner 'DnsSpoofScanner'           [ OK ]
  Id: a30b4de2-84e4-4209-956f-d64376e10101
  Version: 1.00.00.00
Loading handler 'TestHandler'               [ OK ]
  Id: e61374e6-4525-47ca-9fb8-0aa4313c7671
  Version: 1.00.00.00
Opening device 'en0' for capture...          [ OK ]
Starting capture on device 'en0'            [ OK ]
capture: arp 002162FA320F -> 000000000000 (Length: 0)
capture: udp fe80::c0cc:bef:5afe:93a2:62872 -> ff02::1:3:5355 (Length: 43)
capture: tcp 199.47.217.149:80 -> 64.187.249.6:49190 (Length: 179)
nslookup: sjc-not10.sjc.dropbox.com
nslookup: 64.187.249.6
capture: tcp 64.187.249.6:49190 -> 199.47.217.149:80 (Length: 0)
capture: tcp 64.187.249.6:49190 -> 199.47.217.149:80 (Length: 391)
capture: tcp 199.47.217.149:80 -> 64.187.249.6:49190 (Length: 0)
capture: udp 64.187.242.117:50068 -> 255.255.255.255:10019 (Length: 128)
scheduler: running scanner 'Arp Spoof Scanner'
scheduler: running scanner 'DNS Spoof Scanner'
nslookup: 64.187.242.117
nslookup: broadcasthost
```

(Application Startup Sequence)

Wire Sentry provides several command-line options for configuration. These options allow the application to be executed with different options and in different environments. Wire Sentry uses the open-source *NDesk.Options*⁴ library to parse command-line arguments. The command-line arguments are as follows:

- -d: Capture packets on the specified network device.
- -c (optional): Use the specified connection string to store log information in a MySQL database server.
- -v (optional): Increase verbosity of debug messages to the console.
- -n (optional): Do not capture in promiscuous mode, only capture packets destined for this interface.



```
mlaptop:Debug mlandi$ mono WireSentry.exe

-----
| Wire Sentry                               Version 0.9 |
| Copyright © 2013 Michael Landi             |
|                                             |
| mlandi@sourcesecure.net                   |
| http://www.sourcesecure.net               |
|-----|

usage: wsentryd -d {DEVICE} -c {CONNECTION_STRING} -v

-d, --device=DEVICE    the name of the DEVICE to capture
-c=CONNECTION          the database CONNECTION string
-v                    increase debug message verbosity
-n, --normal          don't use promiscuous mode
-h, --help            show this message and exit

mlaptop:Debug mlandi$
```

(Application Command-Line Options)

Packet Processing

SharpPCap exposes a minimalistic interface for capturing packets from the network and interfacing with network devices. *SharpPCap* provides functionality for accessing all of the available interfaces on the system. The *CaptureDeviceList* class's static *Instance* property returns a list of all network interfaces on the system and can be enumerated to find a particular interface via the device name. Wire Sentry implements the following *GetDevice* function to enumerate the interface collection and return a handle to the specified device:

```
protected virtual ICaptureDevice GetDevice(string name)
{
    var devices = CaptureDeviceList.Instance.Where(dev => dev.Name == name);
    if (devices.Count() == 0)
    {
        throw new Exception("The device '" + name + "' was not found.");
    }
    return devices.Single();
}
```

(Enumeration of Capture Devices)

Once a handle to the device has been returned, an event handler (or callback function) must be specified to process each packet as it is captured on the NIC (network interface card). This callback function is a logical function address that is executed on each and every packet capture. The *SharpPCap* interface also provides options for capturing packets in both Normal and Promiscuous modes. Normal capture mode will capture only packets destined for this particular network interface card, while promiscuous capture will collect all packets that reach the NIC including packets destined for other devices. Promiscuous mode must be used to capture all network traffic passed to the interface.


```

/* Attempt to open the specified device for capturing... */
try
{
    //Attempt to open the device interface for capture.
    Debugger.Status("Opening device '" + Device + "' for capture...");

    //Configure device for packet capture.
    device = GetDevice(Device);
    device.OnPacketArrival += HandleOnPacketArrival;

    //Open the device depending on the mode specified.
    if (promiscuous)
    {
        device.Open(DeviceMode.Promiscuous);
    }
    else
    {
        device.Open(DeviceMode.Normal);
    }
}

```

(Callback Function Setup and Promiscuous Mode)

When a packet is captured, a CPU interrupt is triggered and the callback function specified during the *SharpPCap* initialization is called on the application's main thread. Wire Sentry implements a special class, *PacketProcessor* to handle the processing of packets as they are captured off of the NIC. The Packet Processor determines the type of packet (UDP, TCP, ARP, ICMP, etc.) and encapsulates the packet header and payload data into a strongly typed class for storage. This class is the *DataPacket* class, a simple container for storing packet data. The packet is then stored in an in-memory cache for later scanning by scanner modules.

```

/// Handles a packet when it is captured from the...
protected virtual void HandleOnPacketArrival(object sender,
{
    //Determine if the packet is a type of packet that we kn
    var packet = Processor.Process(e.Packet);
    if (packet == null)
    {
        //If we can't handle it just discard the packet.
        return;
    }

    //Enter write lock for the cache (we're going to be addi
    _cachelock.EnterWriteLock();
    Cache.Add(packet);
    _cachelock.ExitWriteLock();

    Debugger.Put(5, packet.ToString());
}

```

(Callback Function Implementation, Packet Processing, and Cache Storage)

The *PacketQueue* is an internal in-memory cache for storing the captured packets. As packets are processed via the *PacketProcessor* class they are encapsulated as *DataPacket* objects and then added to the *PacketQueue*. Each scanner periodically queries this data collection to analyze it for malicious activity. The *PacketCache* is a custom data structure that inherits from the standard *System.Collections.Queue<T>* generic class. It extends the default behavior to place a finite capacity for elements that can be stored in the collection. This upper bound ensures that the system's entire memory cannot be consumed by the application. Old packets are removed to make room for newer packets when the queue is filled to capacity.

```
/// Adds the specified packet to the queue...
public virtual void Add(DataPacket packet)
{
    //If the queue is full dequeue the first element.
    while (Count >= MaxSize)
    {
        PacketQueue.Dequeue();
    }

    //Enqueue the packet.
    PacketQueue.Enqueue(packet);
}
```

(Finite Capacity Added to Queue)

The *PacketCache* instance is accessed by multiple threads throughout the lifetime of the application and must be protected from multi-threaded collisions deadlocks and thread starvation. The .NET/Mono Framework provides several different ways to protect against these issues and allow synchronized thread safe access to objects. To facilitate efficient and safe access from multiple threads Wire Sentry implements reader/writer locks using the *System.Threading.ReaderWriterLockSlim* class. This class allows for both Read and Write locks to be obtained and allows for upgrades from read to write locks in order to maintain safe and efficient access to shared data objects.

Once the *PacketProcessor* has finished processing each packet it attempts to perform a reverse DNS lookup on the source and destination IP addresses in the packet. This allows a domain name to be stored along with the packet information for easier viewing of log information. Because DNS lookups require extra time, a separate background thread and work queue were implemented to efficiently handle reverse DNS lookups. This implementation ensures that the main application thread is not blocked and can continue processing packets as they enter the NIC. The *PacketProcessor* initializes an instance of the *System.Threading.EventWaitHandle* class that provides functionality to block a thread until some work must be performed. Calling the *WaitOne* function forces the thread to halt until an item triggers the thread to wake up. The background thread enters an infinite loop and simply waits until there is work to do. When the background thread is triggered by an incoming packet, the thread dequeues *DataPacket* objects and executes a reverse DNS query for each one. When there are no more items left in the queue, the thread returns to the waiting state.

```
//Create a bBackground thread to handle DNS lookups for packets
new Thread(() => {
    while (true)
    {
        //Wait until an element has entered the work queue.
        WaitHandle.WaitOne();

        lock (DnsLookupQueue)
        {
            /* Continue pulling items out of the Queue while items exist... */
            while (DnsLookupQueue.Count != 0)
            {
                var dpacket = DnsLookupQueue.Dequeue();
                dpacket.DomainSource = Dns.Get(dpacket.IpAddressSource);
                dpacket.DomainDestination = Dns.Get(dpacket.IpAddressDestination);
            }
        }
    }
}).Start();
```

(Reverse DNS Lookup Background Thread and Work Queue)

In order to further speed up reverse DNS lookups, Wire Sentry implements a custom DNS Provider that supports in-memory caching. The *CachedDNSProvider* is capable of caching reverse DNS requests and increases performance on subsequent requests. The provider implements a hash table to provide an association between IP addresses and domain names. This implementation provides consistent and efficient access to cached entries as the number of entries grows.

```
//Lock the hashtable for writing.
_lock.EnterWriteLock();

try
{
    /* Check awhile the thread is locked to ensure another thread... */
    domain = Hashtable[address];
    if (!string.IsNullOrEmpty(domain))
    {
        return domain;
    }

    //Attempt to resolve the ip address using the system's DNS providers.
    var host = Dns.GetHostEntry(address);
    if (host == null || string.IsNullOrEmpty(host.HostName))
    {
        return null;
    }

    Debugger.Put(5, "\tnslookup: " + host.HostName);

    //Add the address and domain name to the cache.
    Hashtable.Add(address, host.HostName);

    return host.HostName;
}
```

(CachedDNSProvider Hashtable and Reverse DNS Lookup)

Dynamic Modules

Wire Sentry provides software developers with the ability to extend the application's functionality through the use of dynamically loaded modules. Two types of modules are available through the software development kit: scanners and handlers. Scanners are modules that analyze network packets for patterns and alert the system when an attack pattern is detected. Handlers are modules that allow custom actions to be taken when an attack is detected (i.e. log the attack to disk, alert administrator via email, etc.).

Classes that extend the *WireSentry.SDK.Scanner* class are automatically loaded upon application startup. Because the class is marked as abstract, inheriting descendent types must implement each and every abstract member defined in the class. The following class prototype has been defined:

```
namespace WireSentry.SDK
{
    public abstract class Scanner
    {
        protected virtual IDebug Debugger { get; set; }

        public Scanner(IDebug debugger)
        {
            Debugger = debugger;
        }

        public abstract int Frequency { get; }

        public abstract Guid Id { get; }

        public abstract string Author { get; }

        public abstract string Name { get; }

        public abstract string Version { get; }

        public abstract IEnumerable<ScannerResult> Scan(IDataPacketCollection packets);
    }
}
```

(Scanner Module Prototype)

Function	Return Type	Description
Frequency	System.Int32	Retrieves the frequency that the scanner should execute in seconds.
Id	System.Guid	Returns the unique identifier of this <i>Scanner</i> . This value is used to distinguish one <i>Scanner</i> from another.
Author	System.String	Returns the author of this <i>Scanner</i> .
Name	System.String	Returns the friendly name of the <i>Scanner</i> .
Version	System.String	Returns the version number of this module (used for debug information).
Scan(IDataPacketCollection)	IEnumerable<ScannerResult>	Scans the collection of data packets and returns a collection of <i>ScannerResult</i> objects which specify attack details.

Classes that extend the *WireSentry.SDK.Handler* class are automatically loaded upon startup. Like the *Scanner* class, the *Handler* class is also abstract and inheriting descendants must implement all abstract members of the class. The class prototype is defined as follows:

```
namespace WireSentry.SDK
{
    public abstract class Handler
    {
        protected virtual IDebug Debugger { get; set; }

        public Handler(IDebug debugger)
        {
            Debugger = debugger;
        }

        public abstract Guid Id { get; }

        public abstract string Author { get; }

        public abstract string Name { get; }

        public abstract string Version { get; }

        public abstract void Handle(ScannerResult results);
    }
}
```

(Handler Module Prototype)

Function	Return Type	Description
Id	System.Guid	Returns the unique identifier of this <i>Scanner</i> . This value is used to distinguish one <i>Scanner</i> from another.
Author	System.String	Returns the author of this <i>Scanner</i> .
Name	System.String	Returns the friendly name of the <i>Scanner</i> .
Version	System.String	Returns the version number of this module (used for debug information).
Handle(ScannerResult)	void	Executes a custom action based on a particular attack indicated in the <i>ScannerResult</i> class.

Wire Sentry dynamically loads all detected modules in the application's path directory. The startup directory is scanned for all binary files with an *.exe or *.dll filename extension using the *System.IO.Directory* class's *GetFiles* function. Software based reflection is then used to load each assembly into memory and search the assembly for any classes which are descendants of the *Scanner* or *Handler* classes. The framework provided *IsSubclassOf* method determines if a class type inherits from another class type. The *ModuleManager* class is responsible for all loading and execution of dynamic modules.

```
//Loop through all of the assembly files in the directory.
foreach (var file in Directory.GetFiles(path, ".*", SearchOption.AllDirectories).Where
    f => f.EndsWith(".exe", StringComparison.CurrentCultureIgnoreCase) ||
        f.EndsWith(".dll", StringComparison.CurrentCultureIgnoreCase)))
{
    try
    {
        /* Load each assembly into memory and attempt to... */
        var assembly = Assembly.LoadFile(file);
        foreach (var type in assembly.GetTypes())
        {
            if (type.IsSubclassOf(typeof(Scanner)))
            {
                //Add this module to the list of scanners to load.
                scannersToLoad.Add(type);
            }

            if (type.IsSubclassOf(typeof(Handler)))
            {
                //Add this module to the list of handlers to load.
                handlersToLoad.Add(type);
            }
        }
    }
    catch { continue; }
}
```

(ModuleManager Dynamically Loading Modules)

Once a module has been loaded, the application will attempt to create a new instance of this class in memory. Because the *ModuleManager* only knows the abstract type of the class it does not have a handle to call a valid class constructor directly. The Mono/.NET Frameworks provide developers with the *System.Activator* class for this very purpose. Calling the *Activator* class's static *CreateInstance* function will attempt to call the default constructor of a particular class type. If successful, the activator will return a handle to the new instance of the class. The newly instantiated module is then added to an in-memory hash table of available modules index by their unique identifiers (specified in the *Id* property of both handlers and scanners).

```
/* Create a new instance of the class and determine... */  
var module = (Scanner)Activator.CreateInstance(type, new[] { Debugger });  
if (ScannerList.ContainsKey(module.Id))  
{  
    throw new Exception("Scanner has already been added!");  
}
```

(Activator Creates new Instance of Dynamically Loaded Type)

Once the *ModuleManager* has loaded all modules, they must be scheduled to run at periodic intervals. A background thread is spawned from the main thread to handle the scheduling and execution of *Scanner* modules. The background thread creates a new instance of the *System.Timer* class and schedules it to periodically check if scanners need to be executed. Based on a scanner's *Frequency* property, the next time a particular scanner should run is calculated and stored along with the *Id* of that particular scanner. Each time the timer's interval has elapsed, each module is checked to determine if it is due to execute.


```

/// Occurs when the scheduler needs to check if a...
protected void SchedulerElapsed(object sender, System.Timers.ElapsedEventArgs e)
{
    lock (ScannerList)
    {
        //Loop through the modules to check if any modules are ready to run.
        foreach (var module in ScannerList.Values)
        {
            //Is it passed time to run the module?
            if (DateTime.Now < ScheduledList[module.Id])
            {
                continue;
            }

            Debugger.Put(4, "\tscheduler: running scanner '{0}'", ConsoleColor.DarkCyan, module.Id);

            /* Notify any callback event handlers that the module... */
            ScannerShouldExecute(module, EventArgs.Empty);
            ScheduledList[module.Id] = DateTime.Now.AddSeconds(module.Frequency);
        }
    }
}

```

(Timer Checks if Scanner Modules is Due to Execute)

If a scanner module is scheduled to execute, the *Timer* executes the module's *Scan* function. The *PacketCache* is temporarily locked using the Read/Write lock to provide the module with exclusive access to the cache. Results returned from the scanner are checked with a list of existing attacks to determine if this attack signature has been previously detected. If the attack already exists, the offending packets are appended to the existing attack. If the attack is new, a new attack entry is added in the results list containing the offending packets. At this point any registered handler modules are called with the results of the scanner modules.

```

//Enter read lock for one of the scanners.
_cacheLock.EnterReadLock();
var results = s.Scan(Cache);
_cacheLock.ExitReadLock();

//See if there are any results from the scanner and print if true.
foreach (var result in results)
{
    /* Don't store duplicate entries for this attack, ... */
    if (Results.ContainsKey(result.Signature))
    {
        //Update existing collection of packets and database
        var existing = Results[result.Signature];
        var pcount = existing.Packets.Count();
        existing.AddPackets(result.Packets);
        if (pcount == existing.Packets.Count())
        {
            //Nothing new at all happened.
            continue;
        }
    }

    //Notify the logging destination that the attack was updated.
    Logger.Update(existing);
}

```

(Module Scan Execution and Result Set Processing)

To ensure that new attacks are differentiated from ongoing, existing attacks a unique attack signature must be calculated. First, the unique identifier of the attack scanner (*Id* property), the attacker's IP address, the victim's IP address, and the type of attack are concatenated into one string. Then a MD5 hashing algorithm is applied to this string in order to produce a fixed length, unique signature for this attack. The formula for calculating a unique signature is as follows:

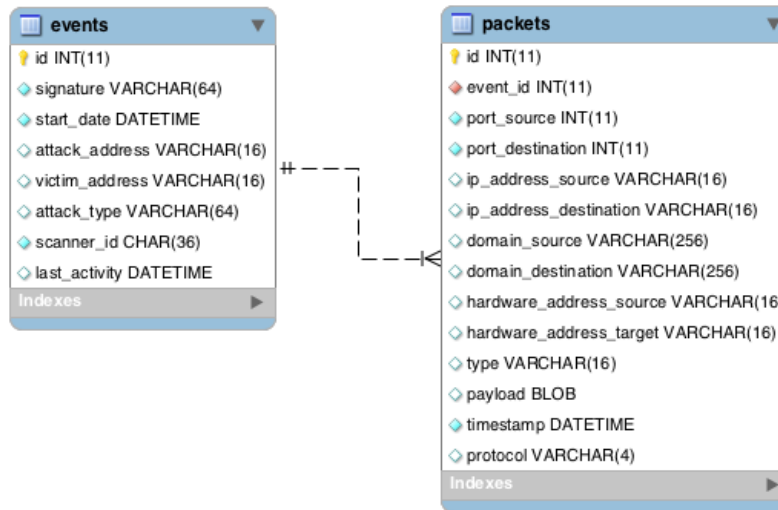
`SIGNATURE = MD5(SCANNER_ID : ATTACKER_IP : VICTIM_IP : ATTACK_TYPE)`

```
/// Calculates an unique signature for the attack...  
protected virtual string CalculateSignature()  
{  
    //Use these variables as input that make the attack unique.  
    var input = Scanner.Id + AttackAddress + VictimAddress + AttackType;  
  
    //Calculate a MD5 hash of the unique input.  
    var builder = new StringBuilder();  
    var bytes = Encoding.ASCII.GetBytes(input);  
    var hash = new MD5CryptoServiceProvider().ComputeHash(bytes);  
    for (int i = 0; i < hash.Length; i++)  
    {  
        builder.Append(hash[i].ToString("X2"));  
    }  
  
    return builder.ToString();  
}
```

(Unique Attack Signature Calculation)

Logging

Wire Sentry has been designed to log attack information into a relational DBMS. By default, Wire Sentry supports MySQL for storage of attack information and related data packets. A simple database schema script is provided along with the source code to create a database that Wire Sentry can store log information to. The schema is as follows:



(MySQL Schema for Attack Log Storage)

SQL queries for insertion and update of records are stored in code in the *MySqlLogger* class. This class provides all of the necessary functionality for connecting to a MySQL database and logging the correct information about each attack. Wire Sentry utilizes the MySQL .NET Connector⁵ library available freely from Oracle to facilitate a connection to the MySQL database. Users can specify the MySQL connection as a command-line parameter when starting Wire Sentry.

```

/* Update the existing event... */
var updateSql = "UPDATE `events` SET `last_activity` = @last_activity;";
var last = result.Packets.OrderBy(x => x.Timestamp).Last().Timestamp;
var update = new MySqlCommand(updateSql, Connection);
update.Parameters.Add("@last_activity", MySqlDbType.DateTime).Value = last;
update.ExecuteNonQuery();

/* Insert each new packet into the database... */
var packetSql = string.Format("INSERT INTO `packets` ({0},{1},{2},{3},{4},{5},{6}, " +
    "{7},{8},{9},{10},{11},{12}) " +
    "VALUES (@{0},{1},{2},{3},{4},{5},{6}, " +
    "@{7},{8},{9},{10},{11},{12});",
    "event_id",
    "port_source",
    "port_destination",
    "ip_address_source",
    "ip_address_destination",
    "domain_source",
    "domain_destination",
    "hardware_address_source",
    "hardware_address_target",
    "type",
    "payload",
    "timestamp",
    "protocol");

```

(SQL Insert Queries Stored in Code)

The screenshot shows a terminal window with the following content:

```

itc@itc_001_lamp:~
mysql> show tables;
+-----+
| Tables in wiresentry |
+-----+
| events                |
| packets               |
+-----+
2 rows in set (0.02 sec)

mysql> select * from events;
+-----+-----+-----+-----+-----+-----+-----+
| id | signature                                     | start_date          | attack_address | victim_address | attack_type | scanner_id |
+-----+-----+-----+-----+-----+-----+-----+
| 8 | 161BEB15917198E5CEF38DAEA2C027E2 | 2012-09-28 15:30:32 | 406C8F35654E | 78843C20BB16 | ARP Spoof | 930b4de2-8424-4209-956d-d64d176e1000 |
| 9 | 161BEB15917198E5CEF38DAEA2C027E2 | 2012-09-28 15:53:12 | 406C8F35654E | 78843C20BB16 | ARP Spoof | 930b4de2-8424-4209-956d-d64d176e1000 |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.02 sec)

mysql>

```

(Attack Data Stored in Database)

Attack Detection

Many basic network attacks can be simulated using freely available attack tools. One such collection of attack tools is known as *dsniff*⁶. The *dsniff* collection provides a variety of utilities for performing LAN based network attacks. The included *arp spoof* utility will launch an ARP spoofing (also known as ARP poisoning) attack on the local network. In the following example the network's default gateway is spoofed to redirect traffic to an attacker.

[illegible]

(Launch ARP Spoof Attack Against 192.168.1.1)

In order for the *arp spoof* utility to successfully poison ARP tables on listening devices it must continuously broadcast malicious information. The malicious information is broadcasted as gratuitous ARP messages (reply messages sent without a request for the information). In the previous example the *arp spoof* utility continuously broadcasted that the default gateway (192.168.1.1) was located at the attacker's MAC address. Using logic from the way that this attack works allows us to create a straightforward and simple algorithm for detecting this type of attack.

1. Filter all packets except ARP packets.
2. Group all packets by the sender's MAC address.
3. Group all packets a second time by the target's MAC address.
4. Filter to groups that contain more than 20 packets in the past 90 seconds.

5. Add any remaining packets to the list of ARP spoofing attacks.

```
public override IEnumerable<ScannerResult> Scan(IDataPacketCollection packets)
{
    //Create a list of results we can add to as we find new attacks.
    var results = new List<ScannerResult>();
    //Determine the time period that we should look back for packets at.
    var lookback = DateTime.Now.AddMinutes(-1).AddSeconds(-30);
    //Group all ARP packets by the sender.
    var arp_source = packets.Items.Where(
        x => x.Protocol == NetworkProtocol.arp).ToLookup(
        x => x.HardwareAddressSource);

    //Loop through each source address.
    foreach (string mac_source in arp_source.Select(x => x.Key))
    {
        //Group all of the sender packets by the target address.
        var arp_source_target = arp_source[mac_source].ToLookup(
            x => x.HardwareAddressTarget);

        //Loop through each target address.
        foreach (var mac_target in arp_source_target.Select(x => x.Key))
        {
            /* Determine if a certain number of attack packets were found... */
            if (arp_source_target[mac_target].Where(x => x.Timestamp >= lookback).Count() >= 20)
            {
                //Store the packets and result in the list for return.
                var packet = arp_source_target[mac_target].First();
                var result = new ScannerResult(packet.HardwareAddressSource,
                    packet.HardwareAddressTarget,
                    "ARP Spoof",
                    this,
                    arp_source_target[mac_target]);

                results.Add(result);
            }
        }
    }

    return results;
}
```

(ARP Spoof Attack Detection Algorithm)

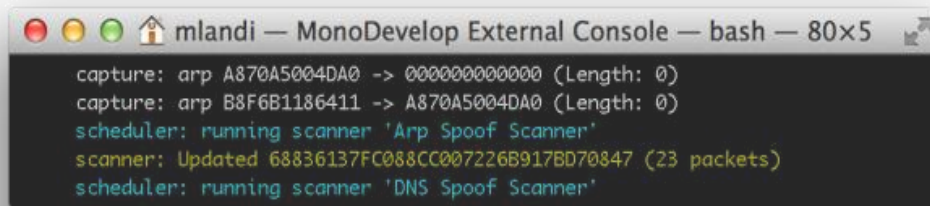
While more advanced, deterministic algorithms can be used in the future to detect ARP spoofing attacks we can see that this simple algorithm can successfully detect these types of attacks. When Wire Sentry detects this attack, a unique signature is calculated, handlers are executed, and the attack signature along with the associated packets are added to the database. Wire Sentry is able to easily detect these types of attacks

```

nslookup: 194.165.188.82
capture: udp 192.168.1.189:64399 -> 157.56.52.12:40022 (Length: 32)
nslookup: 157.56.52.12
capture: udp 192.168.1.189:64399 -> 157.56.52.27:40045 (Length: 37)
nslookup: 157.56.52.27
capture: udp 192.168.1.189:64399 -> 213.199.179.144:40019 (Length: 32)
nslookup: 213.199.179.144
capture: udp 192.168.1.189:64399 -> 213.199.179.160:40010 (Length: 24)
capture: udp 167.206.112.138:53 -> 192.168.1.169:49775 (Length: 181)
capture: udp 167.206.112.138:53 -> 192.168.1.169:50962 (Length: 56)
capture: tcp 192.168.1.189:49210 -> 65.55.71.171:443 (Length: 0)
scheduler: running scanner 'Arp Spoof Scanner'
scanner: 'ARP Spoof' attack B8F6B1186411 -> 2C27D7215543 (20 packets)
Module: Arp Spoof Scanner (1.00.00.38)
Signature: 68836137FC088CC007226B917BD70847
scheduler: running scanner 'DNS Spoof Scanner'
capture: udp 192.168.1.169:60155 -> 167.206.112.138:53 (Length: 45)
capture: udp 167.206.112.138:53 -> 192.168.1.169:60155 (Length: 110)
capture: udp 192.168.1.169:52175 -> 167.206.112.138:53 (Length: 43)
capture: udp 167.206.112.138:53 -> 192.168.1.169:52175 (Length: 111)

```

(New ARP Spoof Attack with 20 Packets)



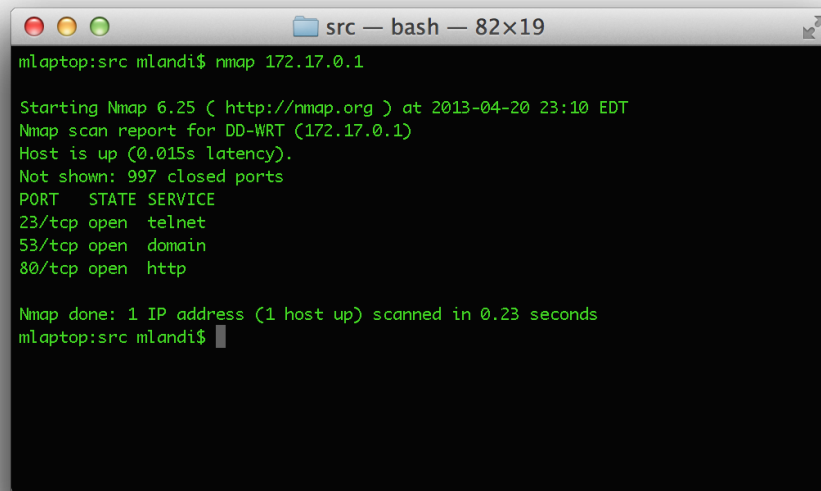
```

mlandi — MonoDevelop External Console — bash — 80x5
capture: arp A870A5004DA0 -> 000000000000 (Length: 0)
capture: arp B8F6B1186411 -> A870A5004DA0 (Length: 0)
scheduler: running scanner 'Arp Spoof Scanner'
scanner: Updated 68836137FC088CC007226B917BD70847 (23 packets)
scheduler: running scanner 'DNS Spoof Scanner'

```

(Updated Ongoing ARP Spoof Attack with 3 Additional Packets)

Another simple type of attack to detect is a port scanning attack. *Nmap*⁷ is an open-source network utility for host detection, port scanning, and operating system fingerprinting. In the following example then network's default gateway is scanned for open ports.

A terminal window titled 'src — bash — 82x19' showing the output of an Nmap scan. The user has entered 'nmap 172.17.0.1'. The output indicates the scan was successful, identifying the host as DD-WRT (172.17.0.1) and listing open ports 23/tcp (telnet), 53/tcp (domain), and 80/tcp (http).

```
mlaptop:src mlandi$ nmap 172.17.0.1

Starting Nmap 6.25 ( http://nmap.org ) at 2013-04-20 23:10 EDT
Nmap scan report for DD-WRT (172.17.0.1)
Host is up (0.015s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE
23/tcp    open  telnet
53/tcp    open  domain
80/tcp    open  http

Nmap done: 1 IP address (1 host up) scanned in 0.23 seconds
mlaptop:src mlandi$
```

(Nmap Port Scanning the Default Gateway)

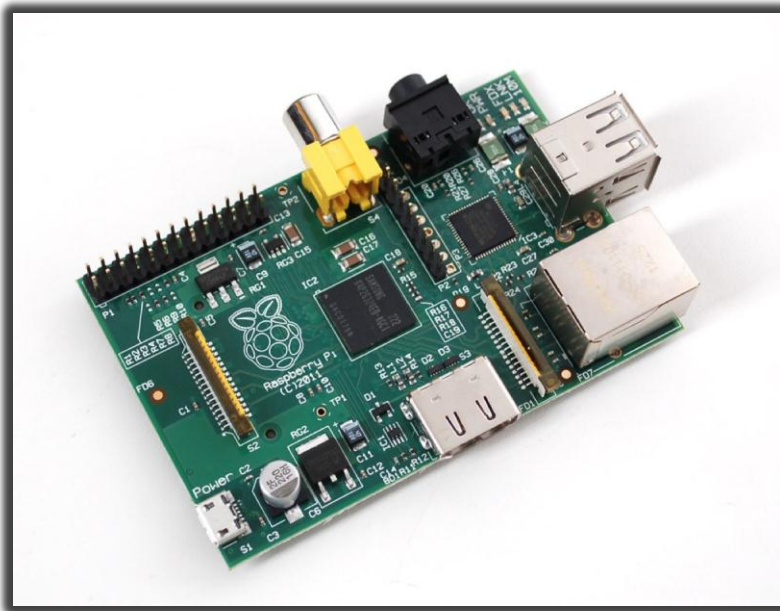
The *Nmap* utility tries to connect to ports on the host in an attempt to determine if that particular port is open for connections. Connections to the ports on the victim's computer generally happen in a randomized order in order to evade some older IDS systems. Through analyzing the way in which these attacks occur we can create an algorithm to detect these types of attacks:

- Filter all packets except TCP/UDP packets.
- Group all packets by the sender's IP address.
- Group all packets a second time by the target's IP address.
- Select all distinct port numbers in the packet results and sort them in ascending order.
- Determine the longest sequence of contiguous port numbers.
- If more than 30 contiguous ports have been accessed in the last 90 seconds add the packets to the list of port scan attacks.

Embedded Hardware

Wire Sentry was designed to be a lightweight intrusion detection framework capable of being placed on cheap, low powered, embedded hardware devices. To accomplish this goal, Wire Sentry was loaded onto a *Raspberry Pi*⁸ computer. The *Raspberry Pi* is a cheap, single-board computer the size of a credit card. The model used in this project had the following specifications:

- \$35.00 Price
- 700 MHz ARM Processor
- 512 MB of Total RAM
- 10/100 MB Ethernet Adapter

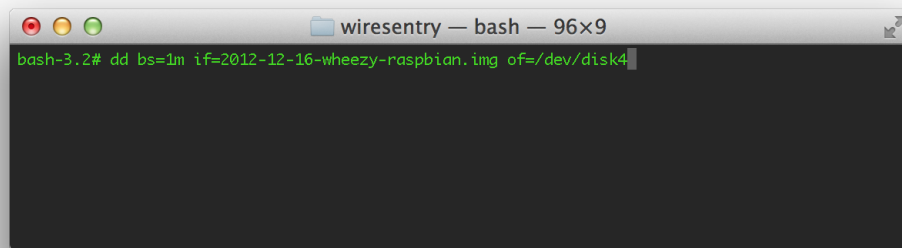


(Raspberry Pi Computer)

The first step to loading Wire Sentry onto the *Raspberry Pi* computer was to select an operating system to use. The recommended operating system is a Debian Linux derivative known as Raspbian⁹. Raspbian is heavily based on the ARM distribution of Debian Linux and has been optimized for the *Raspberry Pi* hardware. Due to some complicated software issues two versions of Raspbian have been released. The

standard version has a faster interface for processing floating-point instructions but cannot support virtual machines such as Oracle JVM or the Mono Framework. The second version is known as “Soft-float Raspbian” uses the older floating-point interface but can support the use of virtual machines. For this reason it is important to select the Soft-float distribution of Raspbian.

Once the Raspbian images were downloaded they needed to be copied to an SD card for loading by the *Raspberry Pi*. Depending on the operating system used, the process of copying an operating system image to an SD card may differ. On UNIX/Linux based systems an easy way to copy the image is to use the *dd* utility.



(Using the dd Utility to Copy Image to SD Card)

Once the image has been copied to the SD card and the SD card is transferred to the *Raspberry Pi*, the *Raspberry Pi* can be powered on and the operating system will boot into a terminal shell. The next important thing to do is install the *libpcap* interface using the automated aptitude package installer. To do this type *sudo apt-get install libpcap0.8* at the terminal.

```

root@raspberrypi:/home/pi# apt-get install libpcap0.8
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnet1
Use 'apt-get autoremove' to remove it.
The following NEW packages will be installed:
  libpcap0.8
0 upgraded, 1 newly installed, 0 to remove and 465 not upgraded.
Need to get 0 B/136 kB of archives.
After this operation, 289 kB of additional disk space will be used.
Selecting previously unselected package libpcap0.8:armel.
(Reading database ... 58754 files and directories currently installed.)
Unpacking libpcap0.8:armel (from .../libpcap0.8_1.3.0-1_armel.deb) ...
Processing triggers for man-db ...
Setting up libpcap0.8:armel (1.3.0-1) ...
root@raspberrypi:/home/pi#

```

(Installing Libpcap Interface)

Once the *libpcap* interface is installed the Mono Framework and Runtime should also be installed using the same method. To install the Mono Framework type *sudo apt-get install mono-complete* at the terminal.

```

root@raspberrypi:~# apt-get install mono-complete
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  libgdiplus libglade2.0-cil libglib2.0-cil libgtk2.0-cil libmono-2.0-1 libmono-2.0-dev
  libmono-accessibility2.0-cil libmono-accessibility4.0-cil libmono-c5-1.1-cil libmono-cairo2.0-cil
  libmono-cairo4.0-cil libmono-cecil-private-cil libmono-cil-dev libmono-codecontracts4.0-cil
  libmono-compilerservices-symbolwriter4.0-cil libmono-corlib2.0-cil libmono-corlib4.0-cil
  libmono-csharpmpg4.0-cil libmono-csharp4.0-cil libmono-custommarshalers4.0-cil
  libmono-data-tds2.0-cil libmono-data-tds4.0-cil libmono-db2-1.0-cil libmono-debugger-soft2.0-cil
  libmono-debugger-soft4.0-cil libmono-http4.0-cil libmono-i18n-cjk4.0-cil libmono-i18n-mideast4.0-cil
  libmono-i18n-other4.0-cil libmono-i18n-rare4.0-cil libmono-i18n-west2.0-cil libmono-i18n-west4.0-cil
  libmono-i18n2.0-cil libmono-i18n4.0-cil libmono-ldap2.0-cil libmono-ldap4.0-cil
  libmono-management2.0-cil libmono-management4.0-cil libmono-messaging-rabbitmq2.0-cil
  libmono-messaging-rabbitmq4.0-cil libmono-messaging2.0-cil libmono-messaging4.0-cil
  libmono-microsoft-build-engine4.0-cil libmono-microsoft-build-framework4.0-cil
  libmono-microsoft-build-tasks-v4.0-4.0-cil libmono-microsoft-build-utilities-v4.0-4.0-cil
  libmono-microsoft-build2.0-cil libmono-microsoft-csharp4.0-cil libmono-microsoft-visualc10.0-cil
  libmono-microsoft-web-infrastructure1.0-cil libmono-microsoft8.0-cil libmono-npgsql2.0-cil
  libmono-npgsql4.0-cil libmono-opensystem-c4.0-cil libmono-oracle2.0-cil libmono-oracle4.0-cil
  libmono-peapi2.0-cil libmono-peapi4.0-cil libmono-posix2.0-cil libmono-posix4.0-cil libmono-profiler
  libmono-rabbitmq2.0-cil libmono-rabbitmq4.0-cil libmono-relaxng2.0-cil libmono-relaxng4.0-cil
  libmono-security2.0-cil libmono-security4.0-cil libmono-sharpzip2.6-cil libmono-sharpzip2.84-cil
  libmono-sharpzip4.84-cil libmono-simd2.0-cil libmono-simd4.0-cil libmono-sqlite2.0-cil
  libmono-sqlite4.0-cil libmono-system-componentmodel-composition4.0-cil
  libmono-system-componentmodel-dataannotations4.0-cil libmono-system-configuration-install4.0-cil
  libmono-system-configuration4.0-cil libmono-system-core4.0-cil
  libmono-system-data-datasetextensions4.0-cil libmono-system-data-linq2.0-cil

```

(Installing Mono Framework)

The last set of packages to install is the *git*¹⁰ source control packages. All Wire Sentry source code and compiled binaries are stored in a freely accessible *git* repository. To automate updating of the software, the current versions can be easily pulled by checking-out the latest version from the *git* repository. To install *git* type *apt-get install git* at the terminal.

```
root@raspberrypi:~# apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  git-daemon-run git-daemon-sysvinit git-doc git-el git-arch git-cvs git-svn git-email git-gui gitk
  gitweb
The following NEW packages will be installed:
  git
0 upgraded, 1 newly installed, 0 to remove and 309 not upgraded.
Need to get 0 B/5,880 kB of archives.
After this operation, 11.5 MB of additional disk space will be used.
Selecting previously unselected package git.
(Reading database ... 58252 files and directories currently installed.)
Unpacking git (from .../git_1%3a1.7.10.4-1+wheezy1_armel.deb) ...
Setting up git (1:1.7.10.4-1+wheezy1) ...
root@raspberrypi:~#
```

(Installing Git)

To pull the latest version of Wire Sentry from the source control the following actions must be taken:

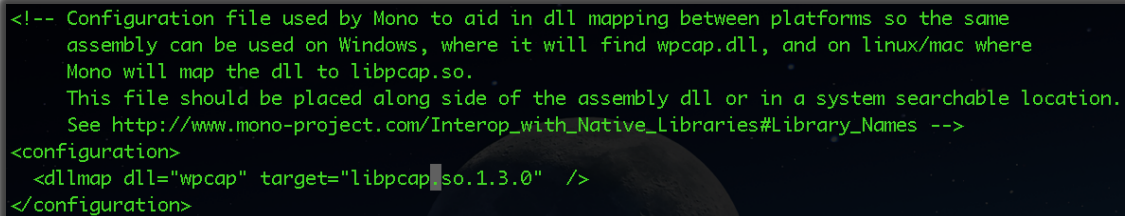
- *mkdir ~/wiresentry* (create a new directory for the application)
- *cd ~/wiresentry* (change directory into the new directory)
- *git init* (initialize this directory as a *git* repository)
- *git pull git@github.com:michaellandi/wiresentry.git* (pull the latest version of source code and binaries).

```
pi@raspberrypi ~ $ mkdir ~/wiresentry
pi@raspberrypi ~ $ cd ~/wiresentry
pi@raspberrypi ~/wiresentry $ git init
Initialized empty Git repository in /home/pi/wiresentry/.git/
pi@raspberrypi ~/wiresentry $ git checkout git@github.com:michaellandi/wiresentry.git
error: pathspec 'git@github.com:michaellandi/wiresentry.git' did not match any file(s) known to git.
pi@raspberrypi ~/wiresentry $ git pull git@github.com:michaellandi/wiresentry.git
remote: Counting objects: 136, done.
remote: Compressing objects: 100% (104/104), done.
remote: Total 136 (delta 16), reused 131 (delta 14)
Receiving objects: 100% (136/136), 492.85 KiB | 669 KiB/s, done.
Resolving deltas: 100% (16/16), done.
From github.com:michaellandi/wiresentry
* branch      HEAD      -> FETCH_HEAD
pi@raspberrypi ~/wiresentry $ ls
data LICENSE.txt README.md src
pi@raspberrypi ~/wiresentry $
```

(Configuring Git Repository)

Once the latest version of the source code and binaries has been pulled from source control we can configure Wire Sentry to run on this particular operating system. The only configuration setting that must be set is the path to the *libpcap* library on the system. For Raspbian the library filename is called *libpcap.so.1.3.0*. This setting must be adjusted in the file located at:

./src/wiresentry/WireSentry/bin/Debug/SharpPCap.ddl.config.



```
<!-- Configuration file used by Mono to aid in dll mapping between platforms so the same
assembly can be used on Windows, where it will find wpcap.dll, and on linux/mac where
Mono will map the dll to libpcap.so.
This file should be placed along side of the assembly dll or in a system searchable location.
See http://www.mono-project.com/Interop_with_Native_Libraries#Library_Names -->
<configuration>
  <dllmap dll="wpcap" target="libpcap.so.1.3.0" />
</configuration>
```

(Modify the Libpcap Library Path and Filename Setting)

Once this setting has been configured correctly, Wire Sentry can be started from the same directory. To start Wire Sentry change the present directory into the *Debug* folder and start the application by typing *mono WireSentry.exe -d eth0*. The application can also be scheduled to start automatically at startup via a *cron* job or other startup script.

Resources

1. Mono Framework: http://www.mono-project.com/Main_Page
2. Libpcap: <http://www.tcpdump.org/>
3. SharpPCap: <http://sourceforge.net/projects/sharppcap/>
4. NDesk.Options: <http://www.ndesk.org/Options>
5. MySQL .NET Connector: <http://dev.mysql.com/downloads/connector/net/>
6. Dsniff: <http://www.monkey.org/~dugsong/dsniff/>
7. Nmap: <http://nmap.org/>
8. Raspberry Pi: <http://www.raspberrypi.org/>
9. Raspbian: <http://www.raspbian.org/>
10. Git: <http://git-scm.com/>