Intravenous AI
Top-level Specification
(The Concept of Operations)
rev. 1b

Mark Grimes
obecian@packetninja.net || obecian@openbsd.org

2nd August 2001

"TCP/IP consists of a suite of protocols, each with their own unique packet structures. RFCs dictate the standards in the form of policy and implementation. A conformity amongst applications at higher levels requires detailed knowledge of underlying network layer definition. Therefore, with knowledge of this universe, we can assert the TCP/IP suite has a finite number of protocol/field combinations that would represent legal moves in order to maintain connection integrity with respect to time. Through the field of Artificial Intelligence, a granular, module-oriented code base can be developed to exploit the knowledge of 'all-plausible moves'."

Abstract

Intravenous (or IV) is a proof-of-concept Intelligent Agent project, that focuses on exploring automated raw socket connections with supplemental attack/defense engines, across a secure Client/Server design. This is essentially, a "distributed packet injection tool on a stick" that can make "intelligent" decisions, utilizing a detailed state table, based on passive connection information gathering, and multi-protocol active discovery.

IV is referred to as an "Agent" throughout this document. An "Agent" is anything (code) that can perceive an *environment* through the use of *sensors* and *effectors*. "Agent" is a frequently used Artificial Intelligence term to describe the imaginary logical human that automates a process that would normally require either a legitimate Client/server connection or extremely fast human intervention.

This is the first whitepaper in a series, outlining the development and case studies from the Intravenous Project. Revisions of this document are restricted to implementation (such as [abstract] datatypes) changes ONLY, to promote maximum optimization. NO changes should ever occur to model state flow.

Project Goals

- OS dependent code-base for dependable APIs
- Active and passive network/host discovery
- TCP/IP header/payload monitoring of in-scope packets
- Client/Server Model
- Extensive TCP/IP network attack infrastructure
- Extensive secure channel communication
- Limited protocol attack discovery
- Limited self-awareness and memory

Design Criteria

The project development of Intravenous involved the commitment of decisions that together have both positive and negative implications. Many conversations transpired after the initial presentation of the draft model at ToorCon `http://www.toorcon.com/`[1] in September 2000. After considering all the constructive criticism in discussions with mathematicians, AI researchers, and network coders alike, I assembled strict guidelines to base all forward logic, for the duration of the project.

Client/Server Model

When IV was first presented, the suggested plan was to create a tool that comprised the entire Agent. Upon execution, the user would be placed in a command-line environment where interactive injection and filtering could take place. The notion was that IV would be OS independent, and the code would be installed on sequentially compromised hosts. The project was initially to use Nemesis `http://www.packetninja.net/nemesis/`[2] to inject packets, with Libnet `http://www.packetfactory.net/Libnet/`[3] and Pcap `http://www.tcpdump.org/` [4] as core dependencies.

The problem with this model is that it fails to provide a solution for many of the project goals. Specifically,

1. Lack of strong cryptography and entropy generation available in some OS's.
2. Limited UNIX flat file host-based awareness.
3. An Agent installed on an insecure (untrusted) OS, is a useless Agent.
4. Lack of correctness in code in some OS's causes unpredictable response.
5. Some OS's violate RFC specifications in either protocol request or replies. Some OS's utilize known reserved fields of protocol headers.
6. The original model limited distributed Agent communication (no point of origin), and introduced multiple points of failure due to problems with Client software possibly afflicting the server (Agent) code.

Clearly, the original project model does not meet the project goals. It relies on third-party software, and intimate knowledge of how each protocol works across each Operating System. Considering, ICMP is perhaps the only intimate research we have, thanks to Ofir Arkin[5], we cannot trust the interaction between each OS kernel and userland (Agent). It is perfectly acceptible to not trust packets arriving from foreign hosts, but it is not acceptible to have to "second guess" each and every packet that comes from each distributed Agent.

To satisfy our need for Agent communication, we turn to the Client/Server infrastructure of network attack. The revised method rolls the Agent into a server daemon. The server will be waiting on a user defined

[1] San Diego security conference

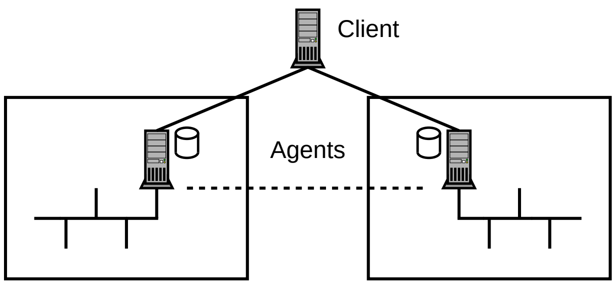
[2] a NINE protocol TCP/IP packet crafting tool-suite

[3] Mike Schiffman's portable packet construction API (http://www.packetfactory.net/Libnet/)

[4] LBNL's portable packet capture library

[5] http://www.sys-security.com/

port for a connection request. The user executes the IV Client, which provides either an interactive or scripted (batch) communication between it and the connected Agents. The Agent challenges upon connection request and further communication occurs across a Blowfish-encrypted, strongly connected TCP tunnel. Instead of a DDoS attack, which is solely positioned to launch a particular Denial of Service attack across the network (among other "master/slave" diagnostic commands), we are providing each "slave" with the ability to launch carefully crafted packets. Anything from detailed IP stack and network based vulnerability, to distributed recon on a firewall on multiple unique interfaces could be plausible.



The dotted line represents the Agent-to-Agent communication. This is an indirect communication that occurs through the "point of origin" (the Client). The proxying of communication (if any) between Agents through the Client, facilitates clean security practice. No communication can be forged Agent-to-Agent or Client-Agent without establishing an authenticated, strongly encrypted TCP connection.

Covert channel communication through "magic packets" can be replayed. If Agent-to-Agent communication has the capability of generating change that taints the Agent's state tables, then that same ability could be employed by an attacker attempting to desynch the Agents, or use them against the Agent deployer.

The goal is the Client makes assertions about tasks for the Agent(s), and the Agent(s) communicates back to the Client for the user to oversee. When used to facilitate an injection that traverses two segments occupied by an Agent in each, it allows for a differential in header information to glean additional data about the network (such as MAC addresses and TTL decrements for enumeration, and analysis of transparent (pass-through) devices)

Two Agents strategically placed on segments where foreign connection endpoints are located, could literally "take the BLIND out of BLIND Spoofing" by presenting two pieces of half duplex information, and a communication channel capable of reassembling a full duplex connection.

Likewise, two Agents placed on different segments could equally engage in ARP spoofing, to increase the amount of traffic sniffed.

Operating System

To warrant both host and network based association inside the Agent, portability must be sacrificed to maximize awareness. The concept of self-awareness cannot be achieved if the Agent has no idea about the world around it, unless there exists a highly restrictive packet capture filter. Access to routing tables and the in-kernel packet filter should be available. With knowledge of both host and network state, an agent can be taught to shunt connections from hostile foreign nodes, or even attack them from multiple agents... indicative of a Distributed Denial of Service attack through "conditional response".

The most important element of the effective Agent, is *TRUST*. If we cannot guarantee that the computer software will execute a task with minimal fault, then the automation becomes counter-productive. Since our Agent has the capacity to launch a devastating attack, sometimes without user-intervention, the model must use an OS that does not compromise security for functionality and/or performance.

Agent trust can be achieved only if it meets two criteria: It works as specified and it is secure. A third element is created to offer the distinction between Agent trust in relationship from a network, or from the quality of outfacing services on a machine. The three goals for a robust model are:

- Security - minimal risk of external compromise (Host Security Trust)

- Crypto - secure communication (Network Security Trust)

- Correctness - the OS works as advertised (Functional Trust)

The OpenBSD Project led by Theo de Raadt, best achieves the goals and attention to detail required of the IV project goals. Solutions to the remaining problems with the original project model that were not addressed by Client/Server design are now fixed by the choice of Operating System.

OpenBSD has not had an external vulnerability discovered in the default install in *over FOUR YEARS*. This provides minimal headache, even though other OS's can be turned into "workstations" to facilitate this same effect. It is not practical to design an Agent that cannot co-exist with other network daemons. The Agent is meant to be used on machines that may be hosting Web, Mail, Name Service, or other common network services.

Due to meeting the above criteria within the base OpenBSD distribution, all dependencies from third-party ports are strictly prohibited. Third party ports are often code designed to run on a multitude of operating systems, and consist of user-land applications and libraries. Many of these libraries, such as Libnet, are abstracted APIs that can increase ease when attempting to write portable code. Since portability is not a concern, dependencies will depend on source code solely maintained by the OpenBSD Development Team.

Last, Software should work as advertised, rather than finding security "silver bullets" that will obfuscate an exploitation attempt. Software vulnerabilities are errors of logic in code, just like other software bugs that may have less critical impact. If a software bug is found, no matter how small, it should be corrected. This seems like proper logic, and is the motivation behind the work of the OpenBSD Development Team.[1]

Declarative versus Procedural

A *declarative* language involves a programmer telling a computer *WHAT* to do, whereas a *procedural* language involves a programmer telling a computer *HOW* to do it. There is not a single AI technique that cannot be implemented by using a procedural language like C.[5]

- C is an extremely popular language that offers a structured skeleton without limiting creativity.
- All APIs utilized in project are also written in C
- AI languages tend to carry many ADTs that may be ill-suited for use for a BSD and RAW socket network tool with minimal AI properties.
- Linking C with LISP is not desirable due to forcing developers to have two learn both languages and manage/coordinate what feels like, two seperate projects.

Threaded Design

A thread is a flow of control within a process. Each thread represents a minimal amount of state; normally just the cpu state and a signal mask. All other process state (such as memory, file descriptors) is shared among all of the threads in the process.

A threaded model will be developed to ensure concurrancy between sensors, expert system and effectors. Each packet that enters the model through the sensor, can be "(next-packet) predicted" and injected, while packets continue to be processed by the sensor.

We will be using OpenBSD *pthreads* (POSIX 1003.1c). John Birrell (jb@freebsd.org) wrote the majority of the user level thread library (-pthread).[3]

Strong Authentication

- In-memory one-way hashes of configuration files (/etc/iv.conf and /etc/ivd.conf)

- Blowfish crypt hash is stored in read-only root-owned (perms 0400) /etc/ivd.conf)

Certainly, password authentication (EKS Blowfish) should be utilized to offer an expensive operation to discourage offline hash cracking, if the Agent node were ever compromised. However, another important issue is the notion of IDENTITY. How do we know for sure that an attacker hasn't cracked an easy dictionary word?

If an attacker cracks the password, the connection will be denied unless the hacker has also compromised the Client host and is sending packets from that IP address. Obviously, alteration of /etc/iv.conf to point to the Client address to that of the hacker, will change the hash, and authentication will be denied (even with legitimate password).

IP address spoofing and the routing of this traffic is outside the scope of this paper. It is assumed that the User will run the Agents on machines configured with security in mind, and that the system security matches that of the Agent's posture. The same rules apply for man-in-the-middle attacks. Lack of security awareness can easily compromise an OS. OpenBSD ships "secure by default", and the Agent implements strategies to prevent both attackers from saturating Agent state and careless user administration of IV configuration files.

Strongly Connected/Encrypted

Cleartext protocols are unacceptable for three reasons:

- Authentication
- Replay Attack
- Connection Hijacking

One would (should) never authenticate across a cleartext protocol, unless a mechanism like S/Key is used to institute one-time passwords. Still, even after authentication, of Telnet (S/Key) (for example), the connection can still be in danger of being hijacked; for instance, when the attacker is sitting on the same segment as either the Client or the Agent. Either way, an attacker sitting in promiscuous mode can do full duplex sniffing of arriving and outgoing packets to a machine on the same segment.

To protect against password discovery and protocol hijacking, the Blowfish block cipher is thrown at the problem. Blowfish is a fast unpatented block cipher designed by Bruce Schneier.[6] It was choosen because it has been "road-tested" and is embedded in the OpenBSD operating system.

Configuration

/etc/iv.conf and /etc/ivd.conf exist on both Client and Agent machines. In order to configure the Agent for operation, these files must be prepared before an IV connection can take place. The user fills in the IP address(es) of the Clients that will be allowed to connect to the Agent. Other data can be placed in comments to decrease likelihood of hash prediction by sniffing incoming Agent connections.

The user then configures /etc/ivd.conf with IP address(es) of the Agents, the exempted ports from Agent inclusion, a Blowfish crypt hash (the password), and followed by Agent metrics for tuning the properties of the Agent (such as state expiration time or probabilistic metrics for hostile code)

> The idea of metrics is to have an attack function tree that runs through static attack procedures with a tunable value from 0.00 to 1.00 as a percentage chance that a particular action will transpire. There is much to be studied in this area of the Agent, but the goal is simply to provide metrics that can allow a user to tweak an Agent's posture - more aggressive or passive, to allow for strategic placement in a dynamic network.

The user "scp's" (secure copies) the completed /etc/iv.conf AND /etc/ivd.conf from the Client to the Agent node through OpenSSH. Now, both the Client and the Agent have copies of the same configuration files.

A second Agent could be stood up by a revision of /etc/ivd.conf with possible changes of exempted ports that match the requirements of that given node (a server farm may be mirrored boxes all installed from the same image – if this is the case, secure copy /etc/ivd.conf to each Agent). The Blowfish hash may be changed to provide a different password for subsequent Agents. Excluded ports are used to prevent the Agent from answering any packet destined for the local Agent on said port(s).

/etc/ivd.conf IS ONLY A TEMPLATE FILE in terms of the CLIENT node.

Once all the files are in place, the Agents need to be started first, and then the Client. The following depicts the Agent configuration files, and their interaction with the Agent program.

- /etc/iv.conf - This is the Client configuration file, with only a few uses.
- /etc/ivd.conf - This is the Agent configuration file, with a vast base of usage.
  (Note: The Client's copy of ivd.conf is intended as template, for creating subsequent Agent ivd.conf's that can be secure copied to the Agent node.)

When the Agent is executed, the following actions take place:

1. /etc/iv[d].conf ownership/permissions are root/0400 or terminate execution.
2. /etc/iv.conf serves as a hash reference point and represents a list of IP addresses that are allowed to connect to the given Agent.
3. List of Agent(s) and exempt ports from /etc/ivd.conf are assembled into a pcap filter.
4. Listen for incoming connections (kqueue!)
5. On incoming connection, password authenticate and compare against Blowfish hash
6. If successful, compute one-way hash of /etc/iv.conf (locally) and challenge the Client.
7. If Client replies with an identical hash, connect the Client to the Agent... otherwise
8. Go to 4 and repeat.

When the Client is executed, the following actions take place:

1. /etc/iv[d].conf ownership/permissions are root/0400 or terminate execution.
2. /etc/iv.conf should be identical to each /etc/iv.conf on the Agent node(s). There is no mechanism slated for "Client Groups". If you let your friend Johnny play with one Agent, then you must let him play with the Agent System (or collection of Agents).
3. Provide User with Front-End (shell environment and batch from command line)
   The command line batch mode will come into play when a rule language for packet injection has been decided.
4. User can perform basic functions such as Agent listings, connects, disconnects, and dumping of statistical data.
5. Connect to Agent and authenticate when prompted with password.
6. Receive challenge from Agent, computer one-way hash of /etc/iv.conf (locally) and respond to the Agent.
7. If connection fails, the user finds him/herself back in shell-like environment, ready to connect the Agent or a different Agent once more.

The password authentication and hash challenge/response provide assurance that even if the password was discovered, the Client IP address in

/etc/iv.conf would have to match that of the attacker PLUS form a hash that will be identical to the hash present on the Agent.

Given the tight controls placed on configuration file ownership, permissions, Blowfish passwords, and doubling identical iv.conf files as a hash-based shared secret; a secret which is not known, but to the superuser of both the Client and Server combined.... this makes for a nice configuration file subsystem that indirectly provides additional function.

Sensors

OpenBSD has both pcap(3) and pthreads(3) as a part of the core distribution, so these are acceptible dependencies. The pcap(3) interface is a high level interface to packet capture.

Intrastate Rules are logically AND'd rules. If *ALL THREE* rules apply, then the packet is accepted into the Intrastate Table (Tree) *WITH STATE*.

AND	Intrastate Rules
IR1	Local Agent Destination
IR2	Protocol Match (bitwise)
IR3	Packet Prediction (Injection State)
IR4	Packet Prediction (Expert System)

Extrastate Rules are logically OR'd rules. If *ANY OF THE FOUR* rules apply, then the packet is accepted into the Extrastate Table (Tree) *WITHOUT STATE*.

OR	Extrastate Rules
ER1	Local Agent Dest (unknown Protocol)
ER2	Local Agent Dest (unknown Source IP)
ER3	Remote Agent Dest (strategic placement)
ER4	Layer 2/3 Broadcast

The term "unknown" refers to the absence of either Protocol or Source IP address given the incoming packets destination is that of the Agent. We conduct a bit-wise compare first, to allow a first-packet for a particular protocol to be accepted by the model with unprecedented speed.

If protocol is found in bitmask, the Intrastate Tree is searched (conveniently, the tree is sorted by Source IP). The list is traversed until a match is found on Extrastate.

Rule Optimization

The unoptimized approach would be to iterate down IR followed by an iteration of ER. If a match occurs, assert it into either Intrastate or Extrastate storage, otherwise drop the packet.

The Intrastate rules are special. IR3/4 actually ties into a portion of the model (the Expert System) that would never get called until AFTER "a packet" (associated with same stream) has already made it into the model by meeting prior Intrastate criteria. In other words, the user would need to inject a packet FIRST, with a (reversed) Injection Table (source address/port) match, and the sniffed packet must have the same protocol-dependent fields that the Inference Engine looks up from the Knowledge Base. This sounds rather confusing, but will make more sense when the Expert System is discussed later in this paper.

So, if an incoming packet already has to have Intrastate in order to satisfy IR3/4, how does such a connection stream become accepted by the Agent? Again, a user-defined injection must take place FIRST to bring "next-packet(s)" into the Intrastate tree (table). The design moved to this method for both functionality and security reasons. By defining Intrastate criteria as such, the chances attackers will corrupt the Intrastate Table are negligible.

Functionally, Intrastate is defined as pertaining to response from a user-defined action. Therefore if the user never decides to send a packet, then the Intrastate tree should contain the empty set (or NULL at the root node).

The security implications of not requiring packet prediction could result in stateful saturation of the Intrastate tree by an attacker. Without IR3/4, an attacker has the potential to initiate state generation in Intrastate, which could have detrimental effects on the performance, not to mention the validity, of Intrastate information. Without IR3, the only requirements are that the packet is being sent to the agent on the same protocol and/or port.

Giving Intrastate priority would seem like the logical thing to do, followed by a Extrastate match, however Intrastate matches (IR1-4) are too intensive for the many more packets that will be solely classified as Extrastate traffic. At the same time, if we reverse this process, there could be traffic patterns that cause an ER match not to occur until ER3 or ER4, which involves a tree search. Given the volume of Extrastate traffic compared to Intrastate, this could be a bottleneck in ensuring Intrastate traffic gets analyzed in a timely fashion.

Initialization

The pcap filter applied is based on data pulled from /etc/ivd.conf. Each Agent address and the local exempted ports make up the basis of the *general* packet filter. However, with such a packet filter employed alone, there is no further data classification. We have already defined the function of the agent (handle 'next-packet' injection), so a method for segregating traffic responses from the rest of the indirectly-related traffic.

This additional classification resembles the rules (IR*/ER*) described above, and are the 'second' filtration that occurs on traffic that passes the general packet filter.

Sensor Flow (with optimization)

Since /etc/ivd.conf contains the list of all Agent IP addresses and the local port exemptions, we can construct a packet capture filter for these basic restrictions.

- Destination IP address == Local Agent
  Destination Port != Local Agent exempted ports
- Destination IP address == Remote IP address

Therefore, assembling a basic pcap[4] filter looks similar to the following:

((ether broadcast) || (ip broadcast) || (ip dst lagent && !eport1 && !eport2 && ... && !eportN) || (ip dst ragent1 || ragent2 || ... || ragent3))

If packets are still being sniffed, given the restrictive packet capture filter, they are valid Intravenous packets and must be categorized as either Intrastate or Extrastate. These classifications determine packet importance, as Intrastate retains complete state and Extrastate does not. To increase the speed in packet classification, we must optimize the IR/ER iterations. The following represents a logical order for rule optimization:

1. If Broadcast (all zeros/ones), mark as Extrastate [ER4]. Otherwise, go to step 2.
2. If packet is destined for remote Agent, mark as Extrastate [ER3]. Otherwise, must be a packet to local Agent. [IR1]
3. A bitwise compare of u_short proto (protocol bitmask). If zero, mark as Extrastate [ER2]. Otherwise we have a match, and proceed to step 4 [IR2].
4. Here, we have ruled out all Extrastate possibilities. The Injection Engine State Table must be analyzed to discern if the packet is in response to a prior injected packet. This is checked by reversing the source/destination IP addresses and ports (if applicable) and then comparing to Injection State [IR3].
5. If match occurs, send packet to Expert System to match protocol dependent relative value changes for 'next-packet' injection [IR4].

Expert System

Expert Systems are primarily of interest due to convenience and availability. They never have to eat, sleep, take holidays, and they always operate at peak performance. They represent the greatest success in AI for commercial systems, due to being generally useful and most importantly unbiased, as it lacks personality. A comparable human expert would have a personality that may conflict with the user. Recommendations from a biased expert, such as religious wars between unices, editors, cryptographic algorithms, become seemingly less important recommendations if the user does not agree. Multiple agents equal multiple experts that will always adhere to the same logic. This creates a cohesive system that can be trusted, where the goal for the Intravenous Expert System is to adhere to TCP/IP RFC documentation.

Every Expert System consists of two major components:

- Knowledge Base - The Knowledge Base represents the solution set.
- Inference Engine - The Inference Engine is used on the Knowledge Base to search for a solution.

Knowledge Bases

Intravenous has three (3) Knowledge Bases that each contain unique information. This information can be divided up into areas of connection integrity, network-based attack and network/host-based defense. Much like a dictionary, the solutions to all defined lookups are contained within.

TCP/IP Knowledge Base

The TCP/IP Knowledge Base contains the specifics for discerning what the 'next-injected' packet needs to contain in order to respond to an Intrastate packet. For example, if the user sends a TCP SYN to a remote host's port 80 (perhaps a client -> web server connection), the TCP/IP Knowledge Base contains the information to respond to the web server's TCP SYN/ACK that will immediately follow.

For example,

- User-defined injection occurs from Agent A to arbitrary Server S on destination port 80.
- S responds to A with a TCP SYN/ACK. (IR1 and IR2 pass, injected TCP has already populated protocol, and packet is being sent to the Agent)
- A sends Injection State packet match to TCP/IP Inference Engine for lookup in the TCP/IP Knowledge Base. A temporary packet is computed with the results from the TCP/IP Knowledge Base. If this is a valid packet, the computed temporary packet will match S's SYN/ACK packet. (This indicates a legitimate response to User-defined injection, and not a spoofed packet. *If it's a spoof, drop the packet!*)
- The next-injected packet is computed against the results from the TCP/IP Knowledge Base and is immediately sent to the Injection Engine for packet injection.

Essentially, the Expert System is used twice per Response/Next-Injection connection pair. The Agent verifies that the incoming packet is a "predicted" packet (to combat IP spoofing), which occurs by computing the relative value changes of the current incoming packet with the prior injected packet. If there is a match, the Agent then computes the relative value changes of the next-injected packet with the current incoming packet and immediately injects. This cycle continues until packet prediction can no longer occur; with hope that the cycle stops because of a completed connection.

Attack Knowledge Base

The Attack Knowledge Base contains high level definitions for network-based attack. A high-level attack is defined as a list of packet combinations required to perform a particular network attack. For example, assigning the injected packet combinations that make up a "smurf" attack or an "arpspoof".

Currently, these methods are function stubs. They will require an attack rules language for defining stateful packets that will be used for high-level attack. The goal is to have an external flat file that can be shared by the Internet community for addition and modification of high-level network-based attack.

Defense Knowledge Base

The Defense Knowledge Base contains high level definitions for *HOST-BASED* defense. This component is largely dependent on the Intravenous code-base understanding the operating system that it is designed for. Currently, the focus of host-based defense has centered on access to dynamic changes made to the OpenBSD PF packet filter[2], along with access to the host's routing tables.

Currently, these methods are function stubs and are not employed in the draft agent. The Defense Knowledge Base is listed in the general specification to define its function and is present for completeness. The PF (based on IPF rules) rule parser will need to be ripped from PF and placed into the agent, so that the User can identify dynamic changes with rules that he/she is already familiar with.

Inference Engines

Intravenous has three (3) inference engines that match their respective knowledge bases. Again, using a dictionary as an example, a Knowledge Base alone, is like owning a dictionary, but not knowing how to look information up. The inference engine is used in examining a packet for knowledge base lookup. The goal of the inference engine is to determine a packet match in as few moves as possible.

There are two primary classifications of an Inference Engine:

- deterministic - predict with *certainty*
- probabilistic - predict with *uncertainty*

A deterministic inference engine makes predictions that are certain, or absolute. Without question, if bytes 13 and 14 of an ethernet header (the protocol field) resemble 0x0806, the protocol is ARP (Address Resolution Protocol). The inference engine can immediately probe the knowledge base for ARP fields and there is no clock cycles wasted on attempting analysis with other protocol headers.

Certainty is defined by RFC. If a network stack is violating RFC, the inference engine will automatically be inaccurate. There will be no face time spent on discovering OS anamolies due to RFC non-compliance - this is counter-productive and will only create complexity on all RFC compliant stacks, forcing the performance to degrade greatly.

There are three ways to construct an inference engine.

- forward-chaining
- backward-chaining
- rule-value

Forward-chaining represents a *data-driven* search that starts with some information and attempts to define an object based on this information. If not enough information is present, the engine requests more information. Backward-chaining is the opposite of this. It starts with an object and tries to verify this object with information. Neither forward-chaining or backward-chaining create an optimal fit given the information provided.

The Rule-Value method is theoretically superior to both forward and backward-chaining, but it can be the most difficult to implement. The Rule-Value method is simply an improved version of backward chaining. The goal of the Rule-Value method is to remove the most uncertainty from the system for each check it makes. This is the optimal method, as not all fields need to be exhausted prior to discovering the hard-fast rule for 'next-packet' injection.

TCP/IP Inference Engine

The TCP/IP Inference Engine is a deterministic engine. There is always some portion of the header that uniquely identifies the packet to a particular major and minor protocol. The major protocol is defined as the overall protocol categorization. Using the above ARP example, detecting 0x0806, defines the major protocol with certainty. By definition, the minor protocol would fall under the major protocol's dependent fields. Therefore, the minor protocol for ARP would be defined by the ARP opcode (bytes 7-8 of the ARP header) - 0x0001 for ARP REQUEST, and 0x0002 for ARP REPLY.

Using this major/minor protocol approach, we create a "rule-value" engine.

Attack Inference Engine

The Attack Inference Engine is a probabilistic engine. There is no real way to discern with certainty that a REMOTE agent is being attacked, due to the local agent containing minimal state of the remote agent(s). For this reason, the engine couples with a configuration metric to determine the probability that a particular attack will be placed on the network wire, given data indicative of network-based attack.

The automated component is planned for preventative measures ONLY. There will be no attacks generated hap-hazardly. The primary reason for establishing the Attack KB/IE pair is to provide a mechanism for user-defined attack. The attack mechanism solely brings current methods of network attack (such as "smurf" and many other network saturation attacks) in-house, working on a packet-by-packet basis (as described in the Knowledge Base section).

Defense Inference Engine

The Defense Inference Engine is a probabilistic engine. There is no real way to discern with certainty that a LOCAL agent is being attacked, due to the variety of ways to affect a node in the TCP/IP environment. A simple spoofed ARP reply to the upstream router, from an attacker sitting on the same LAN as the local agent can disconnect current network traffic.

There are also a variety of both critical and non-critical defensive counter-measures. Criticality is defined as restricting traffic flow through the sensors. An example of a non-critical defensive manuever could be an ARP poison of the legitimate MAC address that the sensor is collecting traffic on. Of course, this is the recommended countermeasure for dealing with attackers using ARP, but the countermeasure also does not affect the flow of traffic - in fact, if anything it re-establishes connectivity and traffic flow that may have been disconnected in an attack.

In contrast, a critical defensive manuever would resemble the closure of firewall rules and/or routes. Either method can impair the sensors traffic flow. The primary concern of critical defensive mechanisms, is that they shall never be performed on Intrastate connection pairs. By definition, Intrastate requires that the User initiates the connection. The design also considers the potential of attackers corrupting the Intrastate tables. Therefore, with these design constraints, there should never be a need to restrict traffic flow on Intrastate traffic - this includes the network interfaces that said Intrastate traffic is passing in and out of.

Although both the Attack and Defense Inference Engines are in their infancy, they are also considered optional engines and are not primary operations that affect the goals of Intravenous.

Injection Engine

The final piece of the model consists of a packet injection engine. It contains two methods of interface: user-defined and automated injection. The Agent institutes STATEFUL PACKET INJECTION. This is a very important concept in both the functionality and security of the model. Without keeping complete state of an injected packet, there is no way for the response to pass IR4 (packet prediction through Expert System). Nothing stops an attacker from crafting a SYN/ACK packet that happens to occur after a User-defined SYN packet.

User-defined Injection

Although the Intelligent Agent is designed so that it can be automated, it is also designed with the notion that a User will provide tasks for the Agent. Without an injected packet, there is no Intrastate traffic, EVER. It is not the goal of this project to invent arbitrary packet injections and maintain a connection across an assertion that is completely random.

At Toorcon, the Intravenous talk centered around the notion of automating injections for the purposes of acquiring missing state information. Of course, this makes a broad assumption that IP spoofing will never match wits with the Agent model. It also makes the assumption that state is kept on ALL packets (Intrastate AND Extrastate). Through experimentation, it was found that the Agent tended to get curious too often, and saturated Intrastate with useless information, as well as created an Extrastate so large that it was time-consuming to parse even in O(log n) time. The model was tuned down in function with security implications kept in mind throughout the entire process.

The new rules that maintain a greater sanity among the data structures follows on the notion that it is impossible to generate Intrastate without the User interfacing with the Agent. Since minimal state (addresses/ports only) is kept on Extrastate traffic, the User can obtain enough information on Extrastate connections (such as some IP address portscanning a remote agent) to assert as a human being, that an injection to the suspect source IP address may warrant bringing the IP into Intrastate, which will maintain complete state. This information can be used for fingerprinting, or network-based attack. As in order for network-based attack to occur, the packet will inevitably be brought into Intrastate as a result of automated packet injection.

The Client is not the component with the injection capability. In fact, the client is simply a front-end to connecting to the Agent, and the Agent is the one with the packet crafting capabililties. Once a connection is stood up between the Client and the Agent, the Client can issue packet injection requests through the Agent. Therefore, the IP address of the Client is hidden from IDS/Network monitors, when injecting.

Since the Agent becomes exposed when performing network-based attack, remote Agents can track incoming packets destined for the Local Agent (via remote Agent Extrastate Tables). However, the only complete state kept on network-based attack is on the packet originator - as this will match the IR rules.

Similar to Nemesis, Intravenous allows for complete crafting of protocol headers with payload from a file descriptor.

Automated Injection

Automated injection ONLY occurs when a User-defined injection precedes it. Hence, a User-defined injection brings a connection pair into Intrastate; once in Intrastate, further injection is handled via automation. The following automated injections occur from results looked up from the TCP/IP Knowledge Base. The injection component, is refered to as an Effector. An Effector acts on the Environment (in this case, Intrastate) as it perceives traffic through the Sensors.

The automated injection component works identical to user-defined injection, with the exception that requests are handled by the Agent from what is learned in the Expert System, and not by User interest.

Conclusion

This whitepaper is meant to serve as an overview of the Intravenous model. The theory as tested (nemesis wrapped with test code) has proved to be a worthwhile experiment. It demonstrates the power of AI structure in typical network code to bring elements of protocol connectivity to methods of TCP/IP that were never intended.

The impact of raw sockets is rather obvious - COMPLETE control of TCP/IP headers. It has been the foundstone of network-based attack, since a majority of network stack weaknesses lie in the arena of the unexpected. Using standard networking (eg. BSD sockets) is quite easy to control because there are accepted standards and functions that have been used for decades to establish connection-oriented client/server design.

Intravenous is a study of emulating standard networking through raw sockets, granting the Agent complete control of packet injection coupled with intelligence in protocol decoding with emphasis placed on maintaining connection integrity with respect to time.

Future whitepapers will break down each component into explicit detail. For now, this specification provides an overview of the project goals and design criteria to establish an understanding of scope and the mechanisms that will be fleshed out with C source code. Stay tuned to my website `http://www.packetninja.net/` for document revisions.

References

[1] T. De Raadt, "OpenBSD", The OpenBSD Project, 1995-2001, Available from http://www.openbsd.org/.

[2] D. Hartmeier, "OpenBSD Packet Filter", The OpenBSD Project, 2001, Available from http://www.openbsd.org/.

[3] C. Provenzano, J. Birrell, "POSIX Threads", The FreeBSD Project, 1995-98, Available from http://www.freebsd.org/.

[4] V. Jacobson, C. Leres and S. McCanne, "libpcap", Lawrence Berkeley National Laboratory, Berkeley, California, 1993-1997.

[5] H. Schildt, *Artificial Intelligence Using C*, McGraw-Hill, Berkeley, California, 1987, pp. 13-15.

[6] B. Schneier, Blowfish Cryptographic Cipher, 1994, Available from http://www.counterpane.com/blowfish.html.