11-2011

The Good, the Bad, and the Actively Verified

John Williamson
Dartmouth College

THE GOOD, THE BAD, AND THE ACTIVELY VERIFIED

Dartmouth Computer Science Technical Report TR2011-710

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

John Williamson

DARTMOUTH COLLEGE

Hanover, New Hampshire

November 2011

Examining Committee:

_____

(chair) Sergey Bratus

_____

Michael Locasto

_____

Sean Smith

_____

Brian W. Pogue, Ph.D.
Dean of Graduate Studies

Abstract

We believe that we can use active probing for compromise recovery. Our intent is to exploit the differences in behavior between compromised and uncompromised systems and use that information to identify those which are not behaving as expected. Those differences may indicate a deviation in either configuration or implementation from what we expect on the network, either of which suggests that the misbehaving entity might not be trustworthy. In this work, we propose and build a case for a method for using altered behavior directly resulting from or introduced as a side-effect of the compromise of a network service to detect the presence of such a compromise. We use several case studies to illustrate our technique, and demonstrate its feasibility with a software tool developed using our method.

Contents

List of Tables

List of Figures

1 Introduction and Motivation

1.1 Problem Statement

Computer networks require trust for normal operation [27], as it allows network entities to accept communication from each other. Compromise in the network breaks that trust until the extent of the damage can be determined. Consequently, full compromise recovery and restoration of network functionality requires reestablishment and verification of that trust [29]. This requirement presents a significant problem in network security, and one that we tackle in our research.

1.2 Trust and its Importance

In a computer network, trust carries a great deal of importance. Trust exists predominently in the form of a three-part relationship: A trusts B for X, where A and B are network entities and X is some service that B provides to A. For example, when a host accepts an offer of a DHCP address, it implicitly trusts the DHCP server it received the offer from. Similarly, when a router participates in an Open Shortest Path First election, it implicitly trusts every other router that is participating. This arrangement exists to save on time and labor of configuration, but can cause problems when the wrong entities are trusted. For example, this trust makes it possible to trick routers into believing false OSPF advertisements [12] [49].

Trust is lost when an element of the network becomes compromised, whether from within or without. Any recovery effort must somehow reestablish this trust. Locasto et al. [29] conducted a study of compromise recovery procedures in computer networks. They concluded that the complexity of networks and attacks on them makes recovery too difficult a problem for a single, comprehensive technical solution. However, they recommend designing recovery tools that have the limitations of human operators as design requirements. The goal of this automation is to help ease the burden on the recovery team and provide them with information that they can easily use in making their decisions. Facilitating compromise recovery in this way motivates our research, and we will revisit it shortly.

1.3 Remote Detection

In our work, we focus on remote detection of compromised network services. Some disadvantages with this approach exist. Chiefly, we acknowledge that it may provide a less comprehensive understanding of what is going on within a system than a more localized approach. However, we find that for our purposes, remote detection can provide enough information, as well as having the advantage of performing all required work from a single or relatively small number of locations. It also eliminates the need to install detection agents throughout the network, which may themselves become untrustworthy.

1.4 Active Attack Observation

The concept of an attack underlies most of network security, yet the term "attack" can refer to many different types of malicious activity. In the absence of cryptographically encoded network traffice, an attack could mean simply sniffing cleartext information from a packet. Modern attacks have evolved, just as networks and their protections have. Many now involve hijacking benign hosts and their network stacks for malicious use, which requires altering the normal, trustworthy behavior of the affected devices. Elements of the network infrastructure, such as routers and switches, are also attractive targets since network hosts frequently trust them implicitly. Compromising them can act as a *force multiplier* for the adversary. Many penetration tool suites, including those of Core Impact [1] and Metasploit [2], take advantage of this implicit trust.

When a compromised machine exists on a network, there are two primary ways to find it. First, we can detect the malicious activity *passively*. The conventional Intrusion Detection System (IDS) paradigm provides a good example, although this approach has a number of well-known shortcomings, as we discuss later in section 2.1.

We focus on a different approach, *active probing*. Active probing attempts to match a known behavioral model with properties observed through responses elicited from a target, and works by

detecting deviations from that model. Those deviations may indicate the presence of If the infection's behavior is known, then a scanner can identify the infection by probes designed to trigger the identified behavior. We make the case that intrusion detection can benefit from active probing. Active probing requires much care, since it may, among other reasons, look like actual malicious activity. Nevertheless, we hypothesize that it can add a significant tool to a sysadmin's arsenel.

Our work takes inspiration from work in active exploit detection, operating system fingerprinting, and vulnerability identification. The Conficker or Downadup worm, unleashed in 2008 and 2009, is one example. The program itself exploited flaws in Microsoft Windows to turn infected machines into components of a worldwide botnet under the control of Conficker's authors [51], prompting excitement in the media [32]. It proved especially difficult to eradicate owing to its level of sophistication, employing a novel combination of several different malware strategies. As a result, the worm proved evasive to conventional techniques [36]. Eventually, active probing proved instrumental in fighting Conficker. Some peer-to-peer strains of the worm used a customized command protocol, which when understood and reverse-engineered, provided a means of scanning for and identifying infected machines [14]. This breakthrough provides a good example of the sort of result we hope for from our own work.

The Zombie Web Server Botnet provides another example of active exploit detection. First documented in September 2009, the exploit targeted machines running web servers, and once installed set up an alternate web server on port 8080. This avoided many simple security measures that only watch port 80. Hidden frames on affected websites contained links pointing to free third-party domain names, which then translated into port 8080 on infected machines. These infected web servers, which also serviced legitimate sites, then attempted to serve malware and other malicious content from this rogue 8080 port onto the redirected user's machine [5]. If the user's web browser did not accept the uploaded malware, the exploit used an HTTP 302 Found status to redirect the user to another infected web server. From there, the exploit re-attempted the

malware upload. This redirection, it turned out, was detectable by sending HTTP GET messages to the queried server and watching for 302 redirects [15].

Let us consider another case, the Energizer DUO USB Battery Charger exploit, documented in March of 2010. The Energizer DUO software, a Windows application which allows users to view the status of charging batteries, installs two `.dll` files: `UsbCharger.dll` in the application directory and `Arucer.dll` in the `C:\WINDOWS\system32` directory. The software itself uses `UsbCharger.dll` to interact with the computer's USB interface, but it also executes `Arucer.dll` and configures it to start automatically (Energizer claimed no knowledge of how this vulnerablity came to be [6]) `Arucer.dll` acts as a Trojan horse, opening an unauthorized backdoor on TCP port 7777 to allow remote users to view directories, send and receive files, and execute programs [19]. While users clearly need to know about this activity, the exploit responds only to outside control, so passive remote detection may not be effective. An active probe, however, can detect the unauthorized port opening even if not in use, and thus identify the infection proactively [16].

Active detection also can detect other sorts of exploitation-related activity, including those taking place out in the network. Consider the common Man in the Middle (MITM) attack. MITM comes in many different varieties, classified according to where they take place (local network, from local to remote, or wholly remote). Once in the middle the attacker has the freedom to act on the packet stream as they please. This opens a number of possibilities, including sniffing data from the packets, connection hijacking, packet injecting, or filtering [38]. Consider DNS hijacking, where an adversary compromises a DNS server or cache. The compromise consists of replacing valid IP addresses with those ones that point to malicious servers, causing the DNS provider to act as a MITM. One technique to detect this involves sending requests to multiple domains, each receiving DNS hosting from a different ISP. Any discrepancy in the responses would indicate that a server or cache has been compromised, which means that MITM is likely occurring [17]. We return to this topic in section 3.3, and draw inspiration from Frias-Martinez et al.'s work in [20], a system for

network access control based on behaviors rather than rules.

Active detection can also work on the infrastructure of the network itself. Most of these examples have focused on network endpoints, but the fabric of the network (routers and switches) can also be exploited and its behavior altered. Consider the famous and controversial presentation by Mike Lynn at BlackHat 2005, in which he highlighted a vulnerability in the Cisco IOS software that could potentially compromise large portions of the Internet's infrastructure [4]. Cisco has attempted to suppress specific details of this research, and widespread exploitation of router vulnerablities still remains unrealized (partially due to the diversity of respective routers' firmware), but we expect that an active probe could detect vulnerable versions of router firmware, should a widespread vulnerability be discovered. If so, this result would be a different example of active detection, scanning for vulnerabilities in a network device rather than existing infections on a host.

1.5 Active Probing Model

We model the behavior of an active probing system as follows. Our model consists of two principals. A, the prober, mimics a protocol stack, and B, the service provider, contains a complete protocol stack. We say that B provides a service X to A, requiring that a trust relationship be established between A and B. For A to consider B trustworthy with respect to service X, B must satisfy a set of constraints C on its behavior. To verify that these constraints hold, A uses a sequence of messages $M = m_1,\ m_2,\ ...,\ m_n$ sent to B, which take the form of packet probes. For each m_i there exists a corresponding message m_i' from B to A in response to message m_i, which may be a packet, a sequence of packets, or the absence of a packet. A may also choose to re-send an unanswered packet as appropriate. For each such m_i', there is some relevant portion $r(m_i')$ that serves as evidence for or against some particular element $c_i \in C$. As each m_i' is received (or not received), A performs the operation $R = R \cup r(m_i')$, building a body of evidence R:

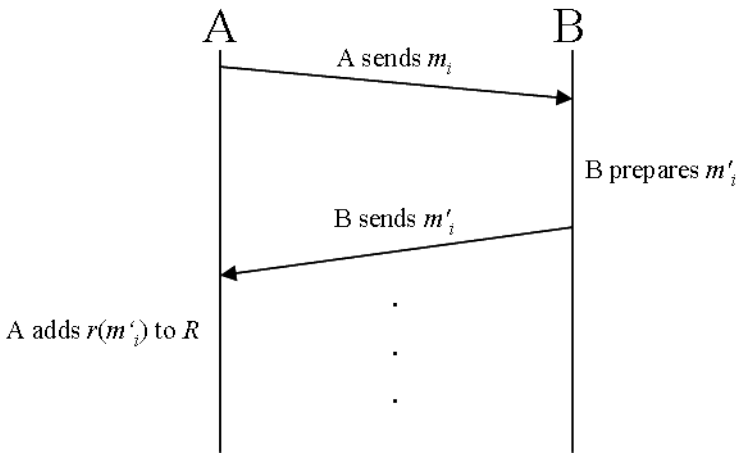

Figure 1: Our active probing data model

Once all probes have been recorded, A makes a determination on whether or not R violates the constraints contained in C, which determines whether or not A's trust in B for service X has been violated. The composition of C and the degree to which R can violate it varies based on the circumstances, so the model does not require anything than that A's trust determination be based on the comparison between R and C.

1.6 Thesis Statement

We believe that we can use active probing for compromise recovery. Our intent is to exploit the differences in behavior between compromised and uncompromised systems and use that information to identify those which are not behaving as expected. Those differences may indicate a deviation in either configuration or implementation from what we expect on the network, either of which suggests that the misbehaving entity might not be trustworthy. In the rest of this work, we propose and build a case for a method for using altered behavior directly resulting from or introduced as a side-effect of the compromise of a network service to detect the presence of such a compromise. We use several case studies to illustrate our technique, and demonstrate its feasibility with a software tool developed using our method.

1.7 Motivation

We believe that the power of active intrusion detection, while recognized, has not yet been fully tapped. Much of the work we found has dealt with detecting specific exploits or vulnerabilities, but we intend something different. Our method is for verifying the trustworthiness of network service providers, and that can be used to raise an alert when verification fails. These active probes will not search for specific exploits, as the examples of section 1.4 do, and thus will not comprise a thorough vulnerability scanner. Instead, they will test for proper functionality. We reason that, in the case of network services, we can determine a set of expected behaviors that indicate trustworthiness of the service provider, so we look at the behavior of the systems we are examining. Our goal is not to find out what causes a problem, but rather whether a potential infection or compromise exists.

We envision our method as applicable to verification of network services which hosts trust by default. We call these services the *Deception Surface* of the network. Since there is usually little or no authentication involved, these protocols present the easiest targets for deception. Consider a few such protocols, along with the trust relationships using the aforementioned notation:

Protocol	**A**	**B**	**X**
DNS	Host	DNS Server	Name resolution
DHCP	Host	DHCP Server	Local network discovery and configuration
ARP	Host/Gateway	Host/Gateway	IP address - MAC Address mapping
STP	Switch	Switch	Switched topology planning
CDP	Network Device	Network Device	Neighbor discovery

Table 1: Example Trust Relationships in Network Protocols

Secure versions of some of these protocols do exist, but they suffer from increased computational overhead. Considering that many network protocols such as DNS employ UDP specifically to reduce overhead and increase speed, this may be considered an unacceptable drawback. It also can

make introducing new entities to the network more difficult, since these must be configured specifically for the new protocols. As a consequence, many networks do not employ services requiring authentication infrastructure to establish basic layer 2 and 3 connectivity.

Hosts trust these protocols by default for labor-saving reasons. For a large enterprise network, configuring each host with authentication credentials for all deception surface network services requires a large investment of valuable time and energy. Most users prefer their machines to work out of the box, and prefer to avoid extensive setup time. For this reason, such authentication frequently does not exist, making deceptions that much more likely. The importance of these services demands trustworthiness, however, so we believe our method for verifying that trustworthiness will prove useful.

One example for which data is available is DNSSEC (DNS Security Extensions), which uses digital signing to prevent tampering with DNS packets. As of June 2011, about 24% of DNS top-level domains are signed [7]. This includes .com and .org, which cover a substantial portion of the existing namespace. Information on what proportion of DNS responses are actually signed, however, is harder. Some surveys have been done of requests, which indicate that at least in certain domains, a significant portion of requests do ask for DNSSEC information, though this tells us very little about whether or not the replies are signed or what is done with them [25]. At this time, it seems that adoption of DNSSEC remains incomplete, and that DNS is a special case, owing to its centralized nature [35]. Other, more distributed network protocols lack this centralized driving force and so are likely to remain less well-secured.

Much work exists on trust relationships in a network, but we found it helpful to view our work in light of Locasto, Greenwald, and Bratus's work [28]. They described a coherent way of depicting the trust relationships in a network, using a visual map of nodess and edges they call *Trust Distribution Diagrams*. Our work here is to check the validity of these trust assumptions using active techniques.

2 Related Work

2.1 Intrusion Detection Systems

A significant portion of existing network defense revolves around the concept of an Intrusion Detection System, or IDS. The IDS concept is applied to protecting both networks and individual hosts, but we focus on the network application. A network IDS generally consists of a detection agent coupled with a sensor, which either stores the data or processes it. The sensor has a set of alert conditions, which raise an alert when triggered. The systems can watch for a wide variety of things, such as traffic anomalies (which require the system to have some concept of normal, either from the programmer or learned through experience), attack signatures (which require knowledge of the attacks being performed), or a combination of these two approaches. Once a detection occurs, the IDS responds, either passively or actively. A passive response involves logging an incident and potentially raising an alert for whatever authority exists on the network. An active response can act on the attacked system to halt or soften the attack, or move against the attacking system to cut off the attacker at the source. Regardless of the response mechanism or other details, the IDS always employs a paradigm of passive monitoring which depends on tracking packet streams and delving into protocols [13]. This leaves them with several fundamental problems, which we will explore.

For an example, consider Bro, a standalone real-time intrusion detection system from the Lawrence Berkeley National Laboratory. Bro emphasizes high-speed monitoring of large amounts of data, prevention of packet loss, real-time result processing, separation of the detection policy and policy enforcement, extensibility, and defensibility. Libpcap underlies Bro, filtering the packet stream based on a set of given criteria (in this case, packets for Finger, Portmapper, FTP, and Telnet connections). An event handler sits on top of the libpcap module and processes the filtered packet streams. This handler may generate and process events if it detects something unusual. A separate handler exists for each event, reinforcing the separation of detection and enforcement. The handlers are scripts that can perform various operations on and with the packet data, generate

more events, create notifications, record data, or modify internal state for subsequent handlers. The end result is an extensible IDS [52].

Network intrusion detection systems like Bro have a number of advantages. First of all, they are passive monitoring systems, which means they do not cause extra load on the network. It also makes them extremely difficult to detect quietly. This means that malicious hosts are not likely to know where exactly the IDS might operate, which makes it difficult to avoid. Strategic sensor placement can make avoiding them all but impossible, since the sensor promiscuously takes packets from the network media. Finally, passive observation can yield a fair amount of information, which can help in detecting certain types of malicious behavior.

In spite of all of these advantages, the IDS paradigm has some inherent flaws, many arising from the IDS's passive nature. The IDS operates by doing detection, raising alerts, and sometimes taking action against offending entities. Consequently, an IDS watches network traffic rather than examining individual machines–their passive nature prevents them from doing so. This limitation makes them unlikely to detect problems unless infected entities are doing something that the IDS deems suspicious. Additionally, watching individual packets often does not suffice to detect suspicious behavior. Generally, some sort of state must be maintained, in such cases as when packets arive out-of-order, and this leads to some new problems. The IDS itself then becomes susceptible to overloading DoS attacks, in which an attacker opens so many bogus connections that the IDS cannot track the actual malicious behavior. Additionally, the IDS may not know what to do with, or it may ignore entirely, connections that were active before it began running. It has no reliable way for establishing context for these connections. The IDS might also face issues with processing speed, where an IDS drops packets from its memory buffer if they arrive faster than they are being processed [43].

The criteria which the IDS uses to make decisions can lead to other issues. A packet in and of itself might cause no harm, but the problems arise once hosts pick up and act on them. Unfortunately,

the IDS can seldom know how every network host will behave for every packet. As a consequence, discerning harmful packets is extremely difficult. Any inconsistency could result in the IDS discarding packets that network hosts might keep (or accepting packets that network hosts might discard). This means those packets either leak through unexamined or the IDS' view of the traffic flow reaching the network becomes distorted. Add to this the difficulty of knowing what other conditions (such as memory overloads or promiscuous NICs) might alter the behavior of hosts, and the problem of knowing how packets will affect the network becomes very difficult to solve [24, 43].

Our probes employ a different paradigm to avoid some of these problems. Consider this list of differences between active and passive paradigms:

Active	**Passive**
Can sound out targets	Must listen to targets
Network overhead	No network overhead
Operates noisily	Operates quietly
Minimal state storage requirement	Potentially significant state requirement
Creates own context	Must learn context from surroundings
Detection based on behavior	Detection based on signature and anomalies
Cannot run constantly	Can run constantly
Cannot run off-line	Can run off-line
Can learn only what is listened for in data model	Can learn everything

Table 2: Difference between Active and Passive Detection Paradigms

We employ *active* monitoring, and as a result we have the ability to systematically probe remote entities to explore their properties. While this approach causes an increase in network load relative to a passive system and may mean that an active scanner cannot run constantly, it also means that the probes have more power and resiliency than a passive monitoring system. We also will need to store less state, making our probes much more difficult to defeat with a DoS than an IDS. This

may arise from the fact that an active monitor does not need to infer connection context the same way a passive one does. Instead, it can create its own context. The fact that an active monitor is only listening when it has taken action, rather than constantly, also helps protect it from DoS-style attacks. Additionally, we do not intend our probes to find specific exploits or vulnerabilities, as an IDS does. Rather, we want to look for behaviors of the system that indicate whether it is in a trustworthy or untrustworthy state. Finally, we set our systems up to learn very specific things based on a specific data model, as opposed to a passive system which can potentially capture *all* information. Recognizing that both approaches have their strengths, we do not propose our system as an alternative to the IDS paradigm, but we seek instead to present a different approach that can supplement what current IDS systems set out to do.

2.2 OS Fingerprinting

We take inspiration from those tools that fingerprint machines and determine their operating system. Active tools to perform this task operate under the same paradigm as the exploits described previously. Just as infected hosts behave differently from well-behaved ones, different systems may react differently to network-based prodding. Active OS fingerprinters use these differences to identify (with some probability, since not all indicators are absolute) the system being used. We also find it interesting to consider systems designed to defeat or deceive this detection, and we will visit those later.

The Nmap network scanner provides one example of an operating system fingerprinter. Nmap contains built-in TCP/IP probes that can both identify the OS and determine other information about it. This information includes device type (router, printer, firewall, etc), an estimate of the system's uptime, an estimate of the next TCP sequence for the host, and the IP ID field sequence for the host. Nmap uses a series of up to 16 crafted probe packets, each of which is crafted for a variation in RFC specifications. Unanswered probes are always re-sent at least once, while information from answered probes gets used to make determinations about the probed machine. The details of this analysis do not concern us in and of themselves, so we do not delve into them.

However, we expect the approach will prove quite useful to us [30].

Now we consider p0f [39], another well-known OS fingerprinting tool. p0f differs from Nmap fundamentally in that it operates using passive detection, and as such has strengths and weaknesses when compared with Nmap. It captures network traffic coming from the remote host and examines it for several common signatures. One is the IP Time To Live field, which different operating systems set differently. Others include the TCP Window Size field, the Don't Fragment IP field, and the Type of Service IP field. Variations in these fields enable p0f and other passive tools to determine what sort of OS sent the packets. Passive systems in general operate more slowly than active systems, and real-time use requires the ability to listen in on traffic from the target host. However, an attacker can rarely detect them, and they can operate off-line in conjunction with other security measures. For example, a system in which an IDS or firewall captures traffic might capture attack packets, and a passive fingerprinting tool like p0f can analyze this traffic and learn more about the attacker.

A different example of passive OS fingerprinting uses DHCP. This method employs a high-level network management protocol, and so operates differently than p0f. David LaPorte and Eric Kollmann presented the idea at Black Hat 2007 [18]. The method proposed uses DHCP Discover, Request, Information, Offer, and Release packets, though Discover, Request, and Information provide the most insight. When there is no DHCP server responding, the job clearly becomes much more difficult, since only Discover packets are heard. Under these circumstances, a scanner can use retransmission timing, though there are some issues when RFC specifications are not followed. The IP Time-to-Live value also serves as a guide to the OS of the sender, though relay agents sometimes reset the value. The preferred method uses the options field, particularly Option 55, the Parameter Request List. The contents, size, and order of this list form a fingerprint that can identify the host's OS. Not only does this give us an interesting result and a good example of OS fingerprinting, but it also deals with a common network service, which we want to look at for our own research. Thus, we look at this system as both inspiration and a reference for our own work.

We now examine Xprobe2, which approaches the problem of what to detect differently. Rather than using the standard signature-matching approach to OS fingerprinting, it employs what its authors call a "fuzzy" approach. They argue that standard signature-matching relies too heavily on volatile specific signature elements. Xprobe2 instead uses a matrix-based fingerprint matching method based on the results of a series of different scans. It scores signatures (out of *No*, *Probably No*, *Probably Yes*, and *Yes*) based on how closely they match the result of each scan, with the highest scoring fingerprint reported as the match. Clearly this gives a significant amount of flexibility in the scoring metrics and scans, which helps allow for things such as modified system behavior, network obstacles, and the like. Xprobe2 is valuable to us in that it illustrates a more sophisticated matching system [11].

Finally, we look at Arkin's ICMP-based network scanning. It details a large number of uses for ICMP in host scanning and attacking, and gives countermeasures for some of them. The work also explores fingerprinting for specific operating systems (Primarily Microsoft Windows 2000). It uses techniques such as watching which systems answer broadcast requests, packets with certain flags set, and so on. All in all, it presents a thorough study of systems' treatment of ICMP, which while useful in itself, also presents an example of an exhaustive protocol analysis. This interests us, as we expect to use this method in our own research [10].

2.3 OS Fingerprinting Deceptions

Another interesting set of tools attempt to fool OS fingerprinters like those we have examined. Consider Provos' work on honeypots [42]. *Honeypot* is a term for a system set up specifically to decoy attackers, hopefully distracting them from real production systems. As a honeypot has no production value to its organization, any attempt to interact with it should raise suspicion. Honeypots exist on the network as either physical machines (with a real computer and IP address) or virtual ones. They might simulate all the functions of the OS, leaving them vulnerable to use by a compromising agent, or only those elements which an attacker cannot use. Physical honeypots have obvious issues with cost and maintenance, so virtual honeypots provide a more practical

solution. This fact makes it even more important for the virtual honeypot to mimic as closely as possible the behavior of the system it poses as. Provos presents a system for virtual honeypots called honeyd, an an effective and efficient means of employing large numbers of virtual honeypots capable of deceiving Nmap and Xprobe.

Another approach is Morph, proposed by Wang at Defcon 12 [26]. Morph acts on signatures of existing production systems, rather than creating decoys. It scrubs and modifyies inbound and outbound traffic to mimic a specific target operating system, fooling both active and passive fingerprinters. The original goals were Microsoft Windows 2000 Service Pack 4, Linux 2.4.x.x, and OpenBSD 3.3. It uses the Packet Purgatory Library, which operates in userland and works between the operating system and network device. It in turn uses the *libpcap* and *libdnet* libraries to interface with the kernel. The Morph system consists of handlers for both inbound and outbound traffic along with a state table. The state table contains session sequence offset information, which tells the handlers how to modify traffic going to a remote OS. Morph can defeat a number of OS fingerprinters, including Nmap, Xprobe, and p0f.

2.4 Other Deceptions

These deceptions have all sought to set up a coherent deception whereby the deceived party believes it is attacking or observing an identifiable system that does not actually exist. That is to say, there is a coherent plan in mind for the deception, and that plan is based on a real system. An alternative defensive approach proposed by Neagoe and Bishop [37] consists of what they call *inconsistent* deception, which seeks to present the attacker with a picture that confuses them, keeping them from knowing what they are looking at and, ideally, making it harder for them to decide what to do.

2.5 Other Work

We also found Enno Rey's work on game theoretic protection of network elements to be useful [48]. The work distinguishes between security through trust and security through control, and suggests

a way of calculating how much to rely on each. We found it an interesting example of a network defense mechanism, and we hope to achieve something similar via a different approach.

2.6 Other Tools

Nmap, the previously discussed network mapper and scanner, has proved useful to us. Most of the time Nmap gathers information about the open ports on a machine, and thus the services it might be offering. However, Nmap now offers the ability to run custom scripts written in the Lua language using their API [21]. This makes Nmap an ideal vehicle for our research, and indeed we have used it for some of our preliminary work. Recent distributions come packaged with a number of code libraries for common network tasks as well as scripts that employ them.

One of these, `sniffer-detect.nse` [31], served as a reference point for some of this preliminary work. This script takes advantage of the fact that a network stack in promiscuous mode will pick up packets that are not intended for it, but after the stack removes the addresses, all higher layers assume the packet is intended for the local stack and act accordingly. The sniffer-detect script uses ARP probes in this manner, and by the responses it hears is able to make a determination about whether or not the probed host is in promiscuous mode. At its core, the approach exploits an assumption made within the stack: that packets which reach the upper layers of the stack are supposed to be answered by that stack. The presence of such assumptions has proven useful to us.

We have also found Scapy [40], a freeware packet manipulation program, quite useful. Scapy allows users to build, sniff, analyze, decode, send, and receive packets with incredible flexibility. It does not interpret the responses directly, so it can be a more powerful tool than Nmap for nonstandard tasks. It employs Python-based control, so its commands are also easily adapted into Python programs. We have used this to conduct much of our research so far, as well as the foundation of our software tool, as Scapy gives us the kind of field-by-field control over packet contents that we require.

We also looked at Yersinia [9] and Loki [3], which are layer 2 and 3 tools designed to manipulate network protocols such as CDP, STP, ARP, BGP, OSPF, EIGRP, and so on. While they did not play a direct role in our research, these tools provided inspiration and examples of how to manipulate and exploit protocols.

3 Active Intrusion Detection Methodology and Approach

In this section, we describe the basic methodology for the detection approach we are proposing.

3.1 Look for Stimulus-Response Pattern

We note that many network interactions take the form of pairing between stimulus and response. The DHCP Discover/Response cycle, the DNS Query/Response cycle, and many others all fall into this category, whereas something like the Cisco Discovery Protocol does not. Note that the stimulus-response includes not only client-server interactions, but also peer-to-peer as well. We rely on and harness this stimulus-response paradigm for our verification method.

We define the participants of our method's scenario based on the A, B, X trust model and active probing model discussed earlier. Our trust relationship consists of two entities, A, the prober, and B, the target, which may be either a host or a server. A trust relationship we can verify or describe with a set of constraints on message content or behavior must exist, with B providing some service X to A. To do this, X requires the exchange of a set of messages between A and B. We need to make determinations about the state of B. To do this, A must send probes to B to build a picture of the state of B, and B must answer them. These answers help build a picture of the system's context.

During our experimentation, we frequently observed that the same stimulus produced different responses from different network entities. We discovered two reasons for this. The first reason relates to configuration. In some cases, responses differ because the two entities operated based on

different configurations. For example, consider two identical DHCP server implementations programmed with different gateways. All other network conditions being equivalent, these two servers will always give a different result when queried, since they are programmed to do so.

The second reason relates to implementation. In most cases, one or more RFCs lay out the behavior a network service or protocol should exhibit. However, in practice we find that differences exist, whether due to lack of specification for every possible case, or simple deviance from the specification. Generally, we found that implementations perform similarly on common cases, such as well-behaved DHCP Discover packets. This observation makes intuitive sense, since specifications exist for them. It is the less well-behaved stimuli that are handled differently.

Taken together, understanding these differences form the foundation of our method. If we look for both types of differences, then two entities must exhibit the exact same behaviors in order to escape notice. Put another way, if an someone wants to masquerade as another on the network, the imitator must mimic not just the target's normal behaviors but the minor, idiosyncratic ones as well.

3.2 Look for Network Trust Relationships and Trusted Data

Trust relationships form the basic building block of the network. As discussed previously, without trust, network operation cannot proceed with any assurance of expectd operation. In the majority of cases, hosts trust essential services by default, to ensure ease of connection without the burden of extensive configuration. As an example, without prior configuration in an IPv4 environment, DHCP and ARP provide the primary ways for a host to learn about the network. Unfortunately, the scope of many modern networks makes these trust-by-default relationships all but necessary, since manually configuring and re-configuring every host in the network is often impractical. As a consequence, they present an avenue for an adversary who can masquerade as a provider of one of these legitimate trusted services. If the adversary offers the same trusted-by-default service and can get his or her information believed, then he or she has compromised whatever elements of the

network believe that information. We target this sort trusted-by-default deception.

Note that we do not assume anything about the exact process by which the deception we have just described is executed. It could be that the adversary has disabled the legitimate service, or is simply able to get its information out faster than the legitimate information does. Regardless of the specifics, we now imagine ourselves in the place of a network entity and interact with a service provider that we assume untrustworthy but whose trustworthiness we must accept for normal operation. Our goal is to verify the trustworthiness of that service provider.

3.3 Employ Cross-Layer Data

Sometimes, it helps to exploit the layered nature of network protocols. Consider the previously discussed man in the middle, one of the most basic and most common compromises. An ordinary machine will pick up all packets and examine them, discarding any that are not addressed to it. This behavior is expected from the majority of well-behaved machines on a network. However, a machine acting as a MITM will pick up these packets, examine them, perform some sort of malicious activity (be it recording, modifying, fuzzing, or any number of other things), and then send them on to their destination. To do this, the attacker must modify the machine's normal network stack, and configure the kernel to forward packets. This modification makes the compromise remotely detectable for us.

Some of our early research dealt with detection of this very forwarding behavior. We hypothesized that if we sent a broadcast packet out to the network with the destination as our own machine, a host configured for forwarding might give itself away by sending the packet back to us. We used Scapy to test this, sending IP packets carrying a layer 2 broadcast address and a layer 3 address of our own machine. We found that many forwarding entities (for example, Linksys routers) did identify themselves by forwarding the packet as expected, but Linux kernels in forwarding mode do not. We hypothesized that this was due to the layer 2 broadcast address of the packet. To test this

hypothesis, we replaced the broadcast hardware address with a unicast address of the machine we wanted to probe, and listened for the response. We found that this resulted in the packet being sent back to us, as expected. We codified this result into an Nmap plugin that detects hosts in forwarding mode, which are behaving in what is generally an undesireable manner and thus may have been compromised or misconfigured.

3.4 Establish Definition of Trustworthiness

In order to establish the trustworthiness of a network service, there must be a notion of what trustworthiness means for that service. This will vary based on the network and service being verified, and in most cases will depend on the specific deployment of the service being probed. For example, trustworthiness in the forwarding case means that no forwarding behavior is exhibited. For a more complicated system, a definition might take into account information the legitimate service should provide, and ways it should respond to certain stimuli. Generally speaking, the definition is what we need to hear to trust the speaker. We will need to use it later on.

3.5 Plan Verification Approach

Once we have an idea of what trustworthiness looks like, we need to develop a plan of how to verify it. Recall the two types of differences between service provders we discussed earlier. Many network entities have peculiarities to their implementations, and the plan for verification should make use of them. It is also necessary to get as much standard information from the probed entity, so that both types of differences can be detected. The more information gathered, both about the service implementation and the service configuration, the harder an adversary must work to fool our probe. In doing this we need to plan to check our service against every part of the trustworthiness definition we have already developed.

3.6 Create Probes

The next step in our methodology calls for turning the plan into a set of active message probes and codifying those probes. The exact mechanism used does not matter, but we found Scapy, the previously described packet generation and manipulation tool, to be helpful. The codified probes comprise the functional portion of an active verification tool, but more still needs to be done.

3.7 Listen for Replies

Finally, we need to capture the replies to our probes and examine them against our trustworthiness definition. With that information, we must make a determination as to the trustworthiness of what we hear.

4 Active Intrusion Detection Meta-Analysis

4.1 Domain Name System

We describe how our method would apply to Domain Name System (DNS). DNS operates on the stimulus-response client-server model, where the client sends name resolution requests to the server, which in turn queries as many other servers in the DNS hierarchy as is necessary to get an answer [33, 34]. In many cases, DNS responses are trusted by default, since they represent the best and frequently only information a host has about how to meaningfully resolve external namespace references. As such, attacks on DNS are fairly common, since successfully doing so could trick a host into sending all of its traffic to the adversary.

Clearly, the trustworthiness of DNS depends on giving correct answers to queries. Our system must be able to determine whether or not the responses it hears are correct. If not, we can assume that the server we are querying is untrustworthy. Note that this does not mean that the server we are querying directly has an issue, but since DNS servers form a hierarchy, a trust issue with one could mean trouble for many others.

For a plan of attack, we clearly cannot examine what answers the server would give for every possible query. We can, however, pick a number of common queries and build a list of responses we should receive for each one. We need a list rather than a single response, as one name frequently has several web servers which respond to queries for it. This list should be large and diverse, and the answers built from manual research or compiled from DNS queries to different servers, minimizing the possibility that a compromised server contributes to our definition of trustworthiness. We also want to feed the server some malformed requests, both with poorly-formed packets and for names known not to exist (this will have to be checked) to test the implementation details of the server.

Our probes would take the form of DNS question packets as described previously, which could be done with scapy. Query responses could be listened for, and responses checked against the list discussed previously. If we hear any unexpected responses, an alert could be raised indicating that a possible issue with the server exists.

4.2 Address Resolution Protocol

We describe how our method would apply to Address Resolution Protocol (ARP). ARP operates on the stimulus-response model where each host or gateway can both make and service requests [41]. ARP provides important information enabling communication both within and across networks, and its information is generally trusted by default, so it provides a good illustration for us.

The definition of trustworthiness for ARP should state that all hosts respond to queries for their IP address with their own MAC address, and gateways also respond to queries for IP addresses outside their network segment with their own MAC address. Any deviance from this model could indicate a deception occurring.

In formulating our trust relationship verification approach, we recognize that only a host can

provide original information about its own MAC address. Consequently, merely looking at ARP replies in isolation may not be sufficient. Consider the following scanning strategy. We conduct an ARP scan of a given set of addresses, and for each address scanned we do two things. First, we listen for replies, and raise an alert if we hear more than one different MAC address in response. Second, if only one response is received, we save it to a hashtable. We then check for one of two things. We allow our scan to either look for the address we have just heard appearing in the hashtable twice, or to look for it to not be in the hashtable.

We need to run this scan against both our own local network (excluding the gateway) and against addresses outside our network. The former warns us of untrustworthy ARP behavior of hosts and servers on our own segment, and the latter of such behavior associated with our gateway. We need the scan to look for both the presence of duplicate addresses and their absence for this reason: all non-local addresses should resolve to the same address, which should not have been seen for any local address. If we do not observe this, we know that we have traffic indended for multiple IP addresses going to the same device on the network. This falls outside our definition of trustworthy ARP behavior, and we raise an alert. We could run forwarding detection against the non-gateway IPs which returned the duplicate MAC, but it is not necessary. Forwarding would indicate a man-in-the-middle occurring, but its absence does not allay our concern.

Our probes take the form of a simple ARP scan, with a supporting hash table. The technique employs a brute-force approach, but should successfully detect ARP issues on the network. Many ARP spoofing detectors are passive, listening only for changes in mappings the detector has heard previously. Active approaches also exist, but may require more than a simple ARP scan [47].

4.3 Border Gateway Protocol

We describe how our method would apply to Border Gateway Protocol (BGP). BGP differs from the previous case studies in that, once in a stable operating state, it operates asynchronously. Once a BGP connection is established between two routers, those routers will periodically

exchange messages. Typical usage occurs between very large networks on the internet, though it can also link different parts of very large private networks as well.

BGP's primary function is to share routing information and make decisions about what routes are best. Our definition of trustworthiness must verify that the route information received is accurate, which we can either determine manually or based on a knowledge of what the router is configured to share, based on how much we know about the router. We must know what the router should tell us before we prompt it to. We also need to get an idea of how often the remote router will send its information without hearing from us, as a change in this could indicate a deception occurring.

Our trust relationship verification approach must begin with opening a BGP connection with a router, since the stimulus-response relationship exists there. To do this we must ensure the router is configured to form a connection with us, and then mimic the behavior of a BGP-enabled router ourselves. We want to deliberately cause the connections to fail and re-initiate new ones, as this produces internal state changes within the remote device that affect its responsiveness. Those states should respond differently to external stimuli, so by probing at each stage we can stress the implementation and determine its behavior carefully. Once a connection is established, we need to listen for the advertisements and compare them against the route information we determined the router should be telling us. We can also tease out the state progression on the remote router by sending or not sending further messages at specific intervals and measuring how often we get responses back, both while we are sending our own keepalives and after we stop. Other configuration details, such as the period of time for which keepalives continue to be sent, should also be tested.

Our probes differ from what we have done before in that not all will immediately produce responses. Rather, some of the messages go towards maintaining up a connection over which configuration data we need will flow. We will need a set of messages for connection establishment, and another set for after the connection is established. We must listen carefully to the responses

we get, and also to their timing, which was less important in the previous cases.

4.4 Discussion

These cases, though the solutions are unimplemented, provide illustrations of how our method is applied. In each case, we start by defining the behavior we deem trustworthy, and then a plan to verify each part of our definition from there. Finally, we describe how our plan would work using active probing. These protocols have some fundamental differences: DNS uses a client-server paradigm, versus a peer-to-peer approach for ARP, and a connection-timer paradigm for BGP. We argue that the fact that our method works for all speaks for its general applicability.

5 Active Intrusion Detection Case Study

5.1 Background

To further demonstrate the viability of this approach, we have implemented WakeUpAndSmellTheServer, an application of our method to the Dynamic Host Configuration Protocol. DHCP makes an excellent subject for a case study for several reasons. First, it provides new hosts several critical pieces of knowledge about the network, such as an IP address, gateway information, and location of the DNS servers. Typically, a network stack sends out a DHCP Discover immediately after coming online, highlighting DHCP's importance. If an adversary can get malicious DHCP information believed, he or she can exert a great deal of control over the deceived hosts. Second, it comprises part of our Deception surface, so most hosts trust whatever DHCP traffic they receive by default.

We see a recent example of an exploit using DHCP in a variant of the Alureon rootkit. This exploit infects networks and sets up a rogue DHCP server to compete with the legitimate one. This rogue server gives out the address of a DNS server under the control of the worm's authors,

which in turn points users to a malicious web server. This web server attempts to force the user to update their browser, but they instead are downloading a malware that will reset their DNS pointer to Google's service once the machine is infected [8]. This is the sort of exploit that motivates us to examine DHCP closely.

The software probes DHCP servers and can both produce `.pcap` fingerprint files and compare to an existing `.pcap` fingerprint. In practice, we found it successfully distinguished between the different servers we used.

5.2 Study Environment

We used three main environments for our DHCP experiments. First, we have a small, self-contained test environment consisting of a pair of computers connected to a single Cisco switch. This was also the environment we used for the aforementioned forwarding detection work. We configured one of the computers with an instance of the udhcpd DHCP server, and ran our tests from the other.

Secondly, we have the actual production network at Dartmouth College's CS department. We used the same machine for our tests here as in the previous environment, and ran our experiments against the actual production server.

Thirdly, we have the DHCP server included in a Linksys WRT54G2 router. It runs as the core of a small home network.

5.3 Definition of Trustworthiness

We thought to assemble a suite of packet probes designed to carefully sound out a server's implementation and configuration. Accordingly, we needed some notion of what trustworthiness means in this scenario, so that we had a way to interpret the results of our probes. We decided to define trustworthiness in terms of a previous state of the server, since a single universally correct

definition may not exist. Practically, this means probing a server at a time when it is assumed to be in a trustworthy state. The trustworthiness of this state can be established in several ways, including deployment, tripwire integrity, and inspection. Once a trustworthy fingerprint is taken, deviance in later scans from the results of that probe indicates a change in the trustworthiness of the server. This principle underlies the structure of the final software product, as we will see.

5.4 Experiment Methodology

As described section 3.1, we tried to determine both the configuration of the probed server and the server itself. We are not interested in identifying the specific server implementation, but rather detecting its differences from a previous set of characteristics that define a model of "trustworthy". We do so by taking a brute-force approach, where we try several different values for different fields. Some of these are well-behaved values, and others are designed to test the server's handling of unusual traffic. This allows us to test both the server's configuration and its implementation.

To test the usefulness of our software, we decided to take a probe fingerprint in one of our environments and check the other environment servers against that fingerprint. Doing so simulates the introduction of another DHCP agent onto the network whose traffic we are seeing instead of the legitimate server's traffic.

5.5 Experiment Tools

This case study required two software components. First, an experimental framework, desined for quick and flexible experimentation. It consists of a python script which employs scapy functionality to send and receive packets. The skeleton script takes a single parameter, the name of a directory which holds two text files. One of these files has the name of the directory with "probe.txt" appended, and the other with "target.txt" appended. The probe file contains, in scapy syntax, the probe packet for the experiment being run. The target file, conversely, contains the target string and sniff count which are fed to a scapy `sniff` function, ensuring that the response to the probe packet is captured. A separate sender thread handles transmission of the probe packet,

allowing the main thread to perform the sniffing for simplicity.

For the actual tool, we wrote up another python program. It also consists of a python skeleton, with the probes and internals in scapy. This program takes two parameters, the name of a target file and an option flag, either `-n` or `-c`. The `-n` flag puts the program into new fingerprint mode, saving the probe responses to a pcap file with the given name. The `-c` flag puts the program into compare mode, comparing its own probe responses to a pcap file given as the other parameter. The program exits gracefully if the two captures are different sizes or the capture specified as the second parameter does not exist. A printout informs the user of any field mismatches between the two capture traces.

5.6 Probes

Consider the structure of a DHCP packet:

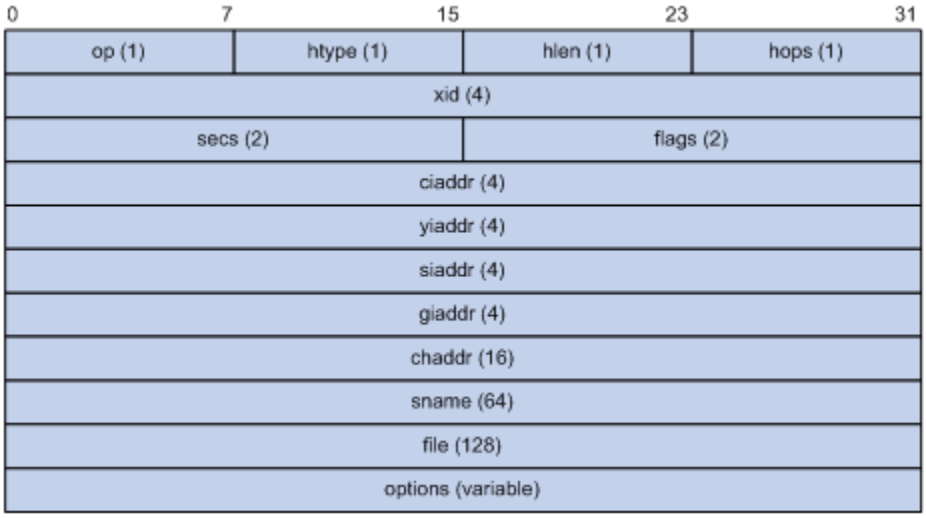

Figure 2: The Structure of a DHCP Packet [23]

Four of these fields (ciaddr, yiaddr, siaddr, and giaddr) contain IP addresses, while a fifth (chaddr) contains a MAC address. We check the servers' handling of these fields by setting each one in turn

to four different types of values:

- The client's currently assigned IP address
- Another valid IP address in the client's subnet
- A valid IP address in another subnet
- An invalid IP address

We also do something similar, though less exhaustive, for the chaddr field. The complete list of probes can be found in Appendix A.

5.7 Results

The tool successfully identified that significant differences exist between the production DHCP server and the Linksys router. Not only were the configurations different, but it turned out that the Linksys DHCP agent in our third environment ignored several of the less well-behaved probes, leading to an easy identification. While not comprehensive, we believe this successful result demonstrates the value of our approach to active intrusion detection.

6 Discussion and Future Work

We discuss here some applications of our method to network security, particularly to the issue of trust recovery.

6.1 Contributions

The chief contribution of our work is a systematic method for developing active packet probing techniques for verifying the trustworthiness of network service providers. That is, for an "A trusts B for X" relationship, we give A a means of verifying that B offers X in a trustworthy manner. We discuss the application of this method to DNS, ARP, and BGP at a high level, and introduce WakeUpAndSmellTheServer, a verification program for DHCP built using our model. We believe

that we have opened the door for significant work in the development of active probing techniques for use in securing modern networks.

6.2 Scanning Application

We envision our method applied to develop a comprehensive active scanner (or an extension of an existing scanner, such as Nmap) which will conduct active probing to do the verification. It needs to be extendable, to accommodate new protocols and necessary. Since our probes work actively, we expect the scanner to operate across broadcast domains, lending flexibility. It can serve two purposes. First, it can routinely scan a network to detect misbehaving entities, such as infected servers, rogue access points, compromised routers, and so on. Doing so could both help to warn of potential problems and detect problems once they actually happen, setting the recovery process in motion. Many networks are routinely scanned by an active system, and our scanner or plugin could act as a part of this. Second, it can help with recovery once an attack takes place, which we hope will be our primary contribution.

6.3 Security Application - Network Trusted Computing Base

Trust in a network differs from trust within individual hosts, but both are related. The concept of a trusted computing base (TCB) comes in key here. According to the well-known *Department of Defense Trusted Computer System Evaluation Criteria* [45], the TCB "contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based." On a system level, this refers to software and data elements of an individual network host, but on a network it gets a bit more complicated.

We want to use the old concept of a *network trusted computing base* or NTCB. The idea of the NTCB, like the TCB, originated with the Department of Defense in the 1980s, this time in *Trusted Network Interpretation* [46]. The NTCB's function parallels that of the TCB, but on a network level. The TCB consists of disparate parts of a computer system which work together to ensure system security. Similarly, security measures on a network can reside on numerous host machines

and take many different forms, including their firewalls, VPN servers, access points, intrusion detection systems, and so on. They perform different functions to ensure the security of the network, such as access control, traffic monitoring, intrusion detection, and auditing. We believe that an active probing system can act as a part of this NTCB, helping ensure that the pieces of the network behave as they should. The NTCB can help the IT staff perform their tasks, especially in the case of a compromise.

6.4 Recovery Application

We envision reestablishing trust after a compromise as the most important application of our research, and the one which originally motivated us. As discussed previously, this trust recovery is a central problem in intrusion recovery. Without some element of trust, recovering from a compromise cannot easily proceed. This problem has received considerable attention for individual hosts, with concepts such as the trusted computing base being the result. One interesting paper on this topic proposes a self-repairing computer system, with the recovery agent and a snapshot of the system serving as the trust base [22]. Other, similar work exists on the individual program level. One paper suggests the use of external software probes to monitor and react to failures in software, which enables the system to recover from errors automatically [50]. Still another suggests rolling back the execution of failed programs to a safe point, and then re-running from there in an environment modified to mitigate the cause of the error [44]. We believe that active probing developed using our method could help establish a system-by-system and service-by-service base of trust on networks, and achieve something similar. As discussed in the previous section, the trusted computing base acts like the portion of the computer system which is responsible for the whole system's security. As part of a trusted network base, active probers could help verify network entities behave as expected, and thus determine on some level which of these entities are trustworthy. This would allow the recovery team to focus their energies and efforts on repairing entities identified as problematic. Clearly, our system will not by itself recover trust, since we are not changing the workings of the probed systems in any way. Instead, it will facilitate trust recovery through active verification of important network protocols.

6.5 Server Masquerade

Our method operated under the assumption that we were masquerading as an ordinary client or peer and eliciting responses from a server or another peer. Another approach might reverse this, and masquerade as a server to interact with clients. This method could help reveal untrustworthy hosts, whereas we have primarily concerned ourselves with untrustworthy servers.

6.6 Self Conversation

Another tweak might involve acting as both the client and server in one, and having a conversation between that client and server that the rest of the network can hear. Such a conversation might occur between layers on the same system, as described in section 3.3. We have not experimented with this, but expect it might be worthwhile to do so in the future.

7 Conclusion

Active intrusion detection stands as an area of great potential in the field of network security. The open nature of many network protocols creates a need for a way to verify their trustworthiness, as otherwise they are blindly trusted. We presented a systematic method to guide the development of trust verification techniques which use active probing as their vehicle of operation. We illustrated how the method would be applied to a couple different open network protocols, DNS and ARP, and implemented a tool using our method for DHCP. The tool achieved its intended purpose when tested in a real network, showing that the method is viable. We believe that our work paves the way for the development of more and robust software tools which can help with trust verification, enhancing the security of modern networks without a significant increase in administrative labor. We also hope that we may aid those charged with recovery from a compromise by verifying that trust should or should not be placed in a particular service. In the future, we envision modifications to our method to address security issues in clients as well as servers and peers. We believe we have contributed to the field of network security through the creation of a novel and general development method for trust verification between network entities.

A DHCP Probes

The following are the probes, in Scapy syntax, used in our testing of WakeUpAndSmellTheServer.

There are several variables used in the probe definition that need to be noted as well:

- `ip`: The IP address of the probing machine
- `otherip`: Another IP address in the same network
- `othernetwork`: An IP address in a different network
- `nonsenseip`: An IP that would never be seen in a production network

```
# Basic Discover probe
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1])/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ ciaddr set to own IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], ciaddr=ip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ ciaddr set to other IP in network
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], ciaddr=otherip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ ciaddr set to out of range IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], ciaddr=othernetwork)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ ciaddr set to nonsense IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], ciaddr=nonsenseip)/
DHCP(options=[("message-type", "discover"), ("end")])
```

```
# Discover probe w/ yiaddr set to own IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], yiaddr=ip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ yiaddr set to other IP in network
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], yiaddr=otherip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ yiaddr set to out of range IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], yiaddr=othernetwork)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ yiaddr set to nonsense IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], yiaddr=nonsenseip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ siaddr set to own IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], siaddr=ip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ siaddr set to other IP in network
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], siaddr=otherip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ siaddr set to out of range IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], siaddr=othernetwork)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ siaddr set to nonsense IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
```

```
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], siaddr=nonsenseip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ giaddr set to own IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], giaddr=ip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ giaddr set to other IP in network
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], giaddr=otherip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ giaddr set to out of range IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], giaddr=othernetwork)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ giaddr set to nonsense IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1], giaddr=nonsenseip)/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe w/ chaddr zeroes
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr="00:00:00:00:00:00")/
DHCP(options=[("message-type", "discover"), ("end")])

# Discover probe with chaddr nonsense
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr="gg:gg:gg:gg:gg")/
DHCP(options=[("message-type", "discover"), ("end")])

# Request probe for own IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1])/
DHCP(options=[("message-type", "request"), ("requested_addr", ip), ("end")])

# Request probe for other IP
```

```
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1])/
DHCP(options=[("message-type", "request"), ("requested_addr", otherip), ("end")])

# Request probe for out of range IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1])/
DHCP(options=[("message-type", "request"), ("requested_addr", othernetwork), ("end")])

# Request probe for nonsense IP
Ether(src=get_if_raw_hwaddr(conf.iface)[1], dst="ff:ff:ff:ff:ff:ff")/
IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68, dport=67)/
BOOTP(flags=0x8000, chaddr=get_if_raw_hwaddr(conf.iface)[1])/
DHCP(options=[("message-type", "request"), ("requested_addr", nonsenseip), ("end")])
```

References

[1] How CORE IMPACT Pro Penetration Testing Works . http://www.coresecurity.com/content/how-it-works.

[2] Learn More about the Metasploit Project. http://www.metasploit.com/learn-more/.

[3] *Loki*. http://ernw.de/content/e6/e180/index_eng.html.

[4] Cisco Acts to Silence Researcher. *BBC News*, July 28, 2005. http://news.bbc.co.uk/2/hi/technology/4724791.stm.

[5] Dynamic DNS and Botnet of Zombie Web Servers. *Unmask Parasites.blog*, September 11, 2009. http://blog.unmaskparasites.com/2009/09/11/dynamic-dns-and-botnet-of-zombie-web-servers/.

[6] "Energizer Announces Duo Charger and USB Charger Software Problem" (Press Release). Energizer Holdings, Inc., March 2010. http://phx.corporate-ir.net/phoenix.zhtml?c=124138&p=irol-newsArticle&ID=1399675&highlight=.

[7] TLD DNSSEC Report (2011-06-14), 2011. http://stats.research.icann.org/dns/tld_report/.

[8] Worm uses built-in DHCP server to spread. *The H: Security News and Open Source Developments*, 2011. http://www.h-online.com/security/news/item/Worm-uses-built-in-DHCP-server-to-spread-1255388.html.

[9] Alfredo Andrés Omella and David Barroso Berrueta. *Yersinia*, 2005. www.yersinia.net.

[10] Ofir Arkin. ICMP Usage in Scanning or Understanding some of the ICMP Protocol's Hazards. Technical report, The Sys-Security Group, December 2000.

[11] Ofir Arkin and Fyodor Yarochkin. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. Technical report, August 2002. http://ofirarkin.files.wordpress.com/2008/11/xprobe2.pdf.

[12] Patrice Auffret and Dror Roecher. OspfAsh, 2007. http://www.gomor.org/bin/view/OspfAsh/ConfItu2007.

[13] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical report, Chalmers University of Technology, 2000.

[14] Ron Bowes. Scanning for Conficker's peer to peer. *Skull Security*, April 25, 2005. http://www.skullsecurity.org/blog/2009/scanning-for-confickers-peer-to-peer.

[15] Ron Bowes. Zombie Web servers: are you one? *Skull Security*, September 11, 2009. http://www.skullsecurity.org/blog/2009/zombie-web-servers-are-you-one.

[16] Ron Bowes. Using nmap to detect the Arucer (ie, Energizer) Trojan. *Skull Security*, March 8, 2010. http://www.skullsecurity.org/blog/2010/using-nmap-to-detect-the-arucer-ie-energizer-trojan.

[17] Jason Coombs. Detecting Man In the Middle Attacks with DNS. *Dr. Dobb's*, December 18, 2003. http://www.drdobbs.com/184416896.

[18] David LaPorte and Eric Kollmann. Using DHCP for Passive OS Identification. BlackHat Japan, 2007.

[19] Will Dormann. Vulnerability Note VU#154421. *US-Cert Vulnerability Notes Database*, March 5, 2005. http://www.kb.cert.org/vuls/id/154421.

[20] V. Frias-Martinez, J. Sherrick, S.J. Stolfo, and A.D. Keromytis. A Network Access Control Mechanism Based on Behavior Profiles. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, December 2009. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5380530.

[21] Gordon Lyon. *Nmap Scripting Engine*. Nmap Reference Guide, Chapter 9, 2010. http://nmap.org/book/nse.html.

[22] Julian B. Grizzard, Sven Krasser, and Henry L. Owen. Towards an Approach for Automatically Repairing Compromised Network Systems. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications*, pages 389–392, 2004.

[23] H3C S3100 Series Ethernet Switches Operation Manual-Release 22XX Series(V1.00). *DHCP Operation*, 2011. http://www.h3c.com/portal/Technical_Support___Documents/Technical_Documents/Switches/H3C_S3100_Series_Switches/Configuration/Operation_Manual/H3C_S3100_OM-Release_22XX_Series%28V1.00%29/201009/690628_1285_0.htm.

[24] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*. USENIX Association, 2001. http://www.usenix.org/events/sec01/full_papers/handley/handley.pdf.

[25] Daniel Karrenberg. DNS Clients Do Request DNSSEC Today. Technical report, RIPE Network Coordination Center, 2010.

[26] Kathy Wang. Frustrating OS Fingerprinting with Morph. In *Proceedings of DEFCON 12*, July 2004. http://www.windowsecurity.com/uplarticle/1/ICMP_Scanning_v2.5.pdf.

[27] Pradip Lamsal. Understanding Trust and Security. Technical report, University of Helsinki, 2001.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.7843&rep=rep1&type=pdf.

[28] Michael Locasto, Steven Greenwald, and Sergey Bratus. Trust Distribution Diagrams: Theory and Applications. In *Proceedings of the Layered Assurance Workshop*, 2010.
http://fm.csl.sri.com/LAW/2010/law2010-06-Locasto.pdf.

[29] Michael E. Locasto, Matthew Burnside, and Darrell Bethea. Pushing Boulders Uphill: the Difficulty of Network Intrusion Recovery. In *Proceedings of the 23rd Conference on Large Installation System Administration*, LISA'09, Berkeley, CA, USA, 2009. USENIX Association.
http://www.usenix.org/event/lisa09/tech/full_papers/locasto.pdf.

[30] Gordon Lyon. *Remote OS Detection*, nmap reference guide, chapter 8 edition, 2010.

[31] Marek Majkowski. sniffer-detect.nse. Nmap Scripting Engine Documentation Portal.
http://nmap.org/nsedoc/scripts/sniffer-detect.html.

[32] John Markoff. Worm Infects Millions of Computers Worldwide. *The New York Times*, January 22, 2009.

[33] P. Mockapetris. RFC 1034: Domain Names - Concepts and Facilities, 1987.
http://www.ietf.org/rfc/rfc1034.txt.

[34] P. Mockapetris. RFC 1035: Domain Names - Implementation and Specification, 1987.
http://www.ietf.org/rfc/rfc1035.txt.

[35] Ram Mohan. DNSSEC Deployment Reaching Critical Mass, 2011.
http://blog.jothan.com/icann/dnssec-deployment-reaching-critical-mass/.

[36] Ben Nahorney. Connecting the dots: Downadup/conficker variants. Technical report, Symantec, 2009.
http://www.symantec.com/connect/blogs/connecting-dots-downadupconficker-variants.

[37] Vicentiu Neagoe and Matt Bishop. Inconsistency in Deception for Defense. In *Proceedings of the New Security Paradigms Workshop*, 2006. http://www.nspw.org/papers/2006/nspw2006-naegoe.pdf.

[38] Alberto Ornaghi and Marco Valleri. Man in the Middle Attacks. In *Proceedings of BlackHat Europe 2003*, 2003. http://alor.antifork.org/talks/MITM-BHeu03.ppt.

[39] Bente Petersen. Intrusion Detection FAQ: What is p0f and what does it do? Technical report, The SANS Institute. http://www.sans.org/security-resources/idfaq/p0f.php.

[40] Philippe Biondi. *Scapy v2.1.1-dev documentation*, April 19, 2010. http://www.secdev.org/projects/scapy/doc/.

[41] David C. Plummer. RFC 826: An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware, 1982. http://tools.ietf.org/html/rfc826.

[42] Niels Provos. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004.

[43] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[44] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, New York, NY, USA, 2005.

[45] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted Computer System Evaluation Criteria. In *National Computer Security Center*, 1985.

[46] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted Network Interpretation. In *National Computer Security Center*, 1987. http://csrc.nist.gov/publications/secpubs/rainbow/tg005.txt.

[47] Vivek Ramachandran and Sukumar Nandi. Detecting ARP Spoofing: An Active Technique. In *Proceedings of the First International Conference on Information Systems Security*, pages 239–250, 2005. http://vivekramachandran.com/docs/arp-spoofing.pdf.

[48] Enno Rey. Games We Play with Our Networks: a Game Theoretic Perspective on Network Configuration and Management. Master's thesis, Danube University Krems, 2009.

[49] Dror Roecher and Patrice Auffret. Routing Protocol Security. In *Proceedings of IT Underground*, 2007. http://www.ernw.de/publikationen/ospf-sec_02_dr.pdf.

[50] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2005. http://dl.acm.org/citation.cfm?id=1247360.1247371.

[51] Gilou Tenebro. W32.Waledac Threat Analysis. Technical report, Symantec, 2009. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/W32_Waledac.pdf.

[52] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association. http://www.usenix.org/publications/library/proceedings/sec98/paxson.html.