Using Active Intrusion Detection to Recover Network Trust

John F. Williamson *Dartmouth College*
Sergey Bratus *Dartmouth College*
Michael E. Locasto *University of Calgary*
Sean W. Smith *Dartmouth College*

Abstract

Most existing intrusion detection systems take a passive approach to observing attacks or noticing exploits. We suggest that *active* intrusion detection (AID) techniques provide value, particularly in scenarios where an administrator attempts to recover a network infrastructure from a compromise. In such cases, an attacker may have corrupted fundamental services (e.g., ARP, DHCP, DNS, NTP), and existing IDS or auditing tools may lack the precision or pervasive deployment to observe symptoms of this corruption. We prototype a specific instance of the active intrusion detection approach: how we can use an AID mechanism based on packet injection to help detect rogue services.

Tags: security, active intrusion detection, networking, trust relationships, recovery

1 Introduction

Existing network intrusion detection systems (e.g., Bro [35, 12], Snort [31]) typically take a passive approach to detecting attacks: they scan network packets and flows to match their content against known-malicious byte patterns (i.e., signatures). Such sensors are typically situated at the network edge or other traffic choke point rather than on individual hosts, and they rarely interpose on (i.e., inject packets or frames into) the actual connection or flow.

IDS systems rarely take an *active* approach to detecting malicious behavior or indicators within the network. By *active*, we mean that the sensor purposefully injects packets and data meant to perturb the state of the network, in essence becoming part of the various connections occurring on the network. Some existing IDS sensors may be "active" in the sense that they periodically scan some hosts or listen to some specific connections, or that they attempt to proactively firewall or quarantine hosts suspected of being malicious (for example, Network Access Control or NAC). To the best of our knowledge, most existing IDSs do not actively participate in network conversations to deduce end host behavior.

This hesitance may be due to the perceived danger of actively issuing network traffic designed to remotely diagnose the existence of malware or corrupted service on an end host or server (such traffic might have an adverse effect on benign hosts or servers).

In this paper, we suggest that the paradigm of active intrusion detection (AID) is relatively under-explored, and we offer an example of how such proactive scanning for malicious behavior at the network level can benefit a system administrator focused on recovering a network infrastructure from an attack that attempts to replace or spoof critical network services.

1.1 Motivation: Intrusion Recovery

Recovering a network infrastructure from an attack — particularly an attack that has compromised a large portion of the infrastructure [19] — is a complex, difficult, and time-consuming task. Furthermore, the administrator may not have much confidence in the services that remain running after the discovery of such a compromise. Because auditing and forensics are expensive processes (in terms of time and density of instrumentation), and such activities can be greatly curtailed because of the need to get the network back up and running, system administrators may have little information about what parts of the system remain trustworthy.

1.2 The Challenge of Recovering Trust

We see the fundamental difficulty in such a situation as the task of *recovering trust* in the network infrastructure. For example, if the DNS server has been compromised, users cannot trust that their DNS queries have not been tampered with. Similar trust relationships exist with ARP and DHCP along with other critical network services. In essence, each protocol implies that the client trusts the server to relay correct information about the network properties. Likewise, the server trusts clients to act only on their own behalf.

Trust exists in many forms within the network. When a host accepts an offer of a DHCP address, it implicitly trusts the DHCP server it received the offer from. Similarly, when a switch participates in a bridge election, it implicitly trusts every other switch that is participating. This arrangement exists out of necessity, but can cause problems when the wrong entities are trusted.

Modern attacks have increased in sophistication; many now involve hijacking benign hosts and their network stacks for malicious use (which requires altering the normal 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 such machines can give an adversary a great deal of power without requiring him or her to attack very many machines. Such attacks are a useful way for an attacker to retain some level of control and spread, and one recent example[1] attempts to run a rogue DHCP service.

Without the ability to meaningfully trust the information such services provide, and in the absence of strong authentication at such low levels of the network (as is typical for very good reasons, see Section 1.4), the task of rebuilding the network from scratch can require a Herculean effort.

In the course of rebuilding trust in critical low-level services, having a tool that can actively probe for the presence of a malicious or compromised low-level service can help identify remnants of an attacker attempting to spoof or man-in-the-middle these services.

1.3 Focus

This paper presents an early step toward a more mature infrastructure for supporting such network recovery activities. Although we are motivated by this problem, our emphasis and focus for the scope of this paper is limited to:

- constructing a data model for representing trust relationships between network services, and;
- implementing a proof-of-concept prototype that uses packet injection (via Scapy[2]) to probe suspect services and examine their responses under the trust relationship model.

These active probes will not search for specific exploits, as the examples we discuss in Section 2.4 do. Thus, we are not aiming to create a thorough vulnerability scanner. Instead, our probes are meant to test for proper functionality and thereby trustworthiness. Our form of active probing is designed less to find out what causes a specific problem than to find whether a potential misconfiguration or malicious influence exists.

1.4 Active Intrusion Detection

Our primary contribution is to propose a new pattern for intrusion detection: actively issuing probes (in the form of specially crafted or purposefully malformed network packets) meant to reveal the presence or operation of rogue services.

Most previous work, even of an active flavor, has dealt with detecting specific vulnerabilities or exploits. In contrast, we introduce a method for crafting active scanning patterns meant to elicit a certain behavior from network hosts. Such a facility can help establish and maintain a basis for verifying the trustworthiness of network services on an ongoing basis (one can think of it as "Tripwire" for network behaviors). We are not searching for specific exploits (as the examples in Section 2.4 do), but rather search for *deception patterns* (i.e., indications that rogue services exist or that otherwise trusted services are compromised).

We believe active probing is most useful in verifying the trustworthiness of certain key network services, including DNS, DHCP, and ARP. We call these services the *Deception Surface* of the network, because it is exactly this fabric upon which most users implicitly (and often unknowingly) base their belief that they are interacting with a trustworthy network connection or service. Since these protocols rarely involve authentication, they are ripe targets for deception.

There are good reasons for *not* employing authentication and authorization infrastructure at such a low network level. The effort involved in managing this equipment and these services in the presence of a variety of different authentication mechanisms and credentials is greatly increased. Without the need to predistribute credentials, hosts are free to "plug-and-play" with the network; being able to simply trust these services by default is a labor-saving practice. 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 (even if a mechanism exists, like DNSSEC) frequently remains unused, thereby leaving room for rogue services and deception.

Central Assumption One of the central assumptions of our approach is the hypothesis that there is an equivalence between "normal" behavior and "trustworthy" behavior. As a consequence, our approach is currently best suited toward detecting malicious influence or attacks that change the normal behavior of a service; in other words, we cannot detect attacks that display syntactically or semantically indistinguishable behavior (and our tool's model of a service's behavior may be incomplete, and thus unable to test features or characteristics that may have been changed). Our tool makes the assumption that changes in normal behavior are symptoms of either malicious influence or misconfiguration.

Our goal with active probing is to significantly raise the bar for an attacker: now they need not only provide a rogue service, but mimic all the logic and failure modes of the "valid" service's code logic and specific configuration. In a sense, active probing helps swing the attacker's traditionally asymmetric advantage to a network defender.

2 Related Work

Our work on active intrusion detection is inspired by recent examples of (largely manual) analysis of the properties and behavior of exploits and malware (see examples below). At the same time, the most related work from a technical perspective is the body of work on OS fingerprinting (see below) and detecting network sniffers (e.g., sniffer-detect.nse [21]). This latter 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. This insight is an excellent independent application of the combination of the Stimulus-Response pattern and the Cross-Layer Data pattern (see Section 3).

Port-knocking is a similar idea to active probing applied to access control: by probing a host with a particular pattern of packets, one can gain the ability to have the target firewall forward subsequent packets. In the theme of verifying host behavior to detect deviations from expected behavior, this work is conceptually related to Frias-Martinez et al. [15], who enable network access control (especially for MANET environments) based on exchanging anomaly detection byte content models.

2.1 Finding Deceptions vs. Monitoring

One central question is how much active probing differs from existing network "good hygiene" monitoring practices like using a second or third independent network connection to actively monitor properties, services, and data that your network exposes to the outside world. We note that active probing is an extension of common practice to proactively scan internal networks with tools like NMap to discover open ports, new machines, or other previously unknown activity at the network edge or within an organization's network core. Rather than just detecting open ports on machines that should not be there, our approach is predicated on reasoning about *deceptions* that exist in the network infrastructure. Although vulnerability scanning software (e.g., NeXpose[3], Nessus[4]) does probe hosts and servers, this type of probing typically focuses on identifying vulnerable versions of software services rather than detecting the presence of malcode or malicious activity.

2.2 Intrusion Detection

Network intrusion detection systems like Snort and Bro [35] have a number of advantages: since they are passive, they do not impose load on the network and they can be difficult to detect. We detail some of the differences between active and passive approaches to IDS in Table 1.

Regardless of the response mechanism or other details, an IDS usually employs a paradigm of passive monitoring which depends on tracking packet streams and delving into protocols [5]. This leaves them with several fundamental problems. IDS, whether passive or active, typically fail-open (i.e., their failure modes do not cease operation of the monitored system and they can not tell when they miss an alert, i.e., false negative).

We suggest that the most relevant shortcoming of current network IDS with respect to the concept of active probing is that an IDS is left to guess the end state of all hosts on the monitored segment. Fundamentally, IDS only observes packet flows and cannot feasibly know the end-state of every host in the network, making it susceptible to evasion attacks [30, 16]. Furthermore, trying to keep track of even limited amounts of state poses a resource exhaustion problem, and even keeping up with certain traffic loads can cause the IDS to miss packets.

2.3 OS Fingerprinting

Nmap uses a series of up to 16 carefully crafted probe packets, each of which is crafted for a variation in RFC specifications [20]. Whereas NMap issues probes to observe the characteristics of the target network stack, the p0f tool uses passive detection, and it examines various protocol fields (e.g., IP TTL, IP Don't Fragment, IP Type of Service, and TCP Window Size) [26]. Alternatively, LaPorte and Kollmann suggest using DHCP for finger-

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 storage
Creates own context	Must learn context from surroundings
Detection based on behavior	Detection based on signature and anomaly
Constant probing is noisy	Can run constantly without disturbing network
Cannot run offline	Can run offline
Can learn only what is listened for in data model	Can learn anything in a trace

Table 1: *A Comparison of Active and Passive IDS Properties*. While both approaches face some of the same challenges (e.g., being fail-open), a hybrid (tightly coupled or otherwise) approach seems promising.

printing [10], and Arkin suggests ICMP [3]. An interesting variation in this field is Xprobe2; rather than using a 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 [4].

Fingerprinting OS network stacks and other services can be an imprecise activity frustrated by the use of virtual honeypots [29] or countermeasures like Wang's Morph (Defcon 12). Morph operates on signatures of existing production systems, rather than creating decoys. Morph scrubs and modifies inbound and outbound traffic to mimic a specific target operating system, fooling both active and passive fingerprinters [18].

2.4 Examples of an Active Pattern

The Conficker worm, unleashed in January 2009, represents one noteworthy example of malware analysis that resulted in a way to diagnose the presence of Conficker's control channel. The malware itself exploited flaws in Microsoft Windows to turn infected machines into a large-scale botnet [22]. It proved especially difficult to eradicate. Because some peer-to-peer strains of the worm used a customized command protocol, subsequent analysis and reverse-engineering provided a means of scanning for and identifying infected machines[6]. This example helps illustrate the utility of the general pattern of active probing for suspect behaviors.

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, thereby avoiding some passive IDS monitors 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 upload malware and other malicious content from this rogue 8080 port [1]. 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 was detectable by sending HTTP GET messages to the queried server and watching for 302 redirects [7].

As a final example of the utility of active probing, consider the Energizer DUO USB Battery Charger exploit (March 2010). The Energizer DUO Windows application allowed users to view the status of charging batteries and installed two `.dll` files, `UsbCharger.dll` in the application directory and `Arucer.dll` in the 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.

`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 [11]. Since this rogue service responds only to outside control, passive detection may not be effective. An active probe, however, can detect the unauthorized open port even if not in use, and thus identify the infection more reliably [8].

2.5 Intrusion Recovery

Recovering a compromised host or network is a difficult task. Classic [34, 33, 9] and more recent [32, 17] accounts can both be found, but little work on systematic approaches to recovery from large scale intrusions exists [14, 25].

3 Approach

When a compromised machine exists on a network, there are two primary ways to find it. First, one can attempt to detect the malicious activity *passively*. Conventional intrusion detection systems provide a good example of this approach. However, compromised or rogue services may not display any behavior that is obviously malicious (thus evading misuse-based sensors) nor display behavior that is particularly new or different than previous packets (thus evading anomaly-based sensors).

Our approach employs *active probing*. The assumption underlying the utility of active probing is that such probing can reveal discrepancies in internal behavior or configuration — particularly at corner cases and for malformed input. In this sense, active probing helps a network defender understand how an infection alters its host's behavior or how rogue services operate.

Active probing is a suitable tool for discovery of latent or otherwise stealthy malicious influence; we can probe hosts (or the network at large using broadcast addresses) rather than waiting for them to send packets. Active probing can constructively infer network state and context by issuing targeted probes.

Active probing exploits several unique features about a networked environment; in essence, this environment represents a distributed state and a set of computations (i.e., the network stacks) involved in manipulating the global state of the network. The arrangement of these relationships and the nature of most protocol interactions provide several key areas of focus for designing probe patterns (e.g., sequences of protocol messages intended to elicit distinguishing responses).

3.1 Key Insight: Behavior Differences Due to Implementation or Configuration

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 richness of the configuration space can help distinguish between a rogue server set up for minimal interposition on a service and the full-featured service.

The second reason relates to **implementation**. In most cases, one or more RFCs lay out the behavior a network service or protocol should exhibit. In practice, however, 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. Corner cases and malformed input (e.g., semantically invalid options pairings or flag settings) cause different, infrequently exercised code paths to execute – it is unlikely that an attacker has replicated such behavior with high fidelity.

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 someone wants to masquerade as another on the network, the imitator must mimic not just the target's normal behaviors in common cases (relatively easy) but the minor, idiosyncratic ones as well (we claim that this is harder).

3.2 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.

3.3 Network Trust Relationships and Trusted Data

Trust relationships form the basic building block of the network. 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 of "trusted–by–default" deception.

Note that we do not specify 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 begin in the place of a network entity and mistrust the service provider that we hear but whose trustworthiness we must accept for normal operation. Everything we do constitutes an attempt to verify the trustworthiness of that service provider. Is the information they provide consistent? Do they respond in the way a legitimate service might if we make illogical or semantically invalid requests? Or, if they are an attacker intent on remaining stealthy, do they greedily respond to packets that look attractive to intercept and interpret, but are really meaningless (in terms of us getting on the network) and only mean to flush out such malicious interposition?

3.4 Cross-Layer Data

Sometimes, it helps to exploit the layered nature of network protocols. Consider a man in the middle attack, 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 detectable (see Section 6 for our experiment on this topic).

4 Active Probing Model

We model active probing on the concept of a network conversation containing messages that reveal the violation of conditions related to configuration or behavior, where these constraints represent the belief of the probing entity about the valid, trustworthy state of the network.

In essence, active probes attempt to verify some behavior of the target host or service, and the messages emitted from the target host in response to our (crafted) protocol messages represent characteristics of that behavior. Figure 1 depicts this interplay in a very basic form; the intent behind probing is to discover behavioral artifacts arising from differences in *implementation* or *configuration* (as discussed in Section 3.1).

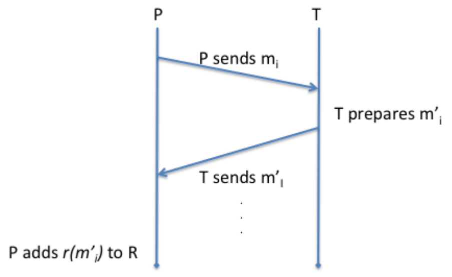

Figure 1: *Ladder Diagram for Active Probing Data Model.* A probing host P (our prototype plays this role) issues probes designed to exercise logic and configuration corner cases in the target host T. As T reacts to these probes (and generates $m_i^{'}$ subject to its implementation quirks and configuration details), P builds a set of data relevant to the trust relationship being probed.

Our model consists of two parties P and T. P is the prober and has the ability to simulate multiple protocol stack implementations (especially "broken" ones). The second, T, is the target or service provider. P's hypothesis is that T may contain a broken, partial, incomplete, or incorrectly configured protocol stack. If T were a trustworthy service, it would display "normal" expected behavior according to the trust relationship between P (rather, the role of the client or peer that P is playing) and T (more specifically, the server or peer that T may be masquerading as). In this sense of having an established trust relationship, we say that T provides a service X to P.

For P to consider T trustworthy with respect to service X, T must satisfy a set of conditions C on its behavior. To verify that these constraints hold, P uses a sequences of messages $M = m_1, m_2, \ldots, m_n$ sent to T that take the form of packet probes.

For each m_i, there exists a corresponding message $m_i^{'}$ from T to P that may be a packet, a sequence of packets, or the absence of a packet (determined through a preconfigured timeout). 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), P performs the operation $R = R \cup r(m_i^{'})$, building a body of evidence R as shown in Figure 1. Once all probes have been sent and answers recorded, the probing entity decides whether or not R violates the conditions contained in C.

5 Methodology

In attempting to recover trust in key network services, a system administrator would follow the general tasks outlined below. The procedures we describe are typically aimed at an auditing-style service rather than a general-purpose scanner for running malware or botnet command and control. As such, the definition of a trust relationship, the specific verification plan, and the format and content of probes are usually service specific and informed by administrator knowledge of their own service implementations and configuration. We posit (but have not shown) that the amount of effort needed to follow the steps below is similar to tuning of IDS rules or calibration of IDS parameters to a specific environment.

5.1 Define 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. This trustworthiness criteria directly informs the set of constraints C.

For example, trustworthiness in the forwarding case means that no hosts but known gateways should exhibit forwarding behavior. For a more complicated system, a definition might take into account information the legitimate service should provide (for example, a known-valid set of DNS responses), and ways it should respond to certain stimuli (e.g., how the service handles a particular corner case configuration or incompatible flags). Generally speaking, the definition is what we need to hear to trust the speaker.

5.2 Verification Plan

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 providers 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.

For the examples above, the verification plan might range from a simple comparison of known good answers to specific DNS queries to the absence of a "forwarded" packet. In essence, each verification plan is tightly coupled to the actual method of detecting a specific deception on the network. As yet, we do not contend with automating this process.

5.3 Probe Creation

The next step in our methodology calls for turning the plan into a set of active message probes and codifying those probes. Although a variety of packet crafting mechanisms exist, we found Scapy, a packet generation and manipulation tool, to be helpful. The codified probes crafted in Scapy's environment comprise the functional portion of an active verification tool.

5.4 Reply Detection

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

5.5 Implementation

We have found Scapy [27], 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 response packets directly, so it can prove more useful than other packet injection or scanning tools in some scenarios. It employs Python-based control, so its commands are also easily adapted into Python programs. We have used Scapy to implement our prototype probing tool. Currently, verification plans (and their corresponding probes) require individually-developed Scapy scripts.

6 Case Studies: Detecting Deceptions

Our preliminary evaluation focuses on illustrating our prototype's effectiveness at detecting network deceptions rather than attempting to detect malicious software (e.g., botnet command-and-control, spyware). To a certain extent, the related work we discuss in Section 2 illustrates how one might go about using existing tools like nmap to identify command-and-control or backdoors. Although we illustrate how to detect (1) a duplicate DHCP server and (2) the presence of a host configured for forwarding, our point is that these two examples are patterns of network deceptions, and this is the main intent of our approach.

6.1 Detecting Forwarding Behavior

As an example of using the Cross-Layer Data pattern, one of our first experiments dealt with detection of forwarding behavior. As ordinary machine will typically silently discard packets (frames) not addressed to it. This behavior is expected from the majority of well-behaved machines on a network. A machine acting as a MITM, however, 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 accomplish this MITM, the attacker must modify the machine's normal network stack settings and configure the kernel to forward packets. This modification is remotely detectable.

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 that are behaving in what is generally an undesirable manner and thus *may* have been compromised or misconfigured.

Formally speaking, in this experiment of detecting a host in forwarding mode, the condition C is that only a small known set of hosts on the network should be in forwarding mode (specifically: that only those hosts should deliver the packet we generated back to us because we chose the packet contents in such a way as to be consumed by hosts that are promiscuous and forwarding, but when processed by higher layers of the network stack, don't realize that they shouldn't be sending this packet back to its origin); if the responses that P gathers contains an IP address outside this set (i.e., we see our message from m_i in R), we know that the trust condition is violated.

6.2 Rogue DHCP Server

To demonstrate the viability of an active probing approach, we have implemented it on 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.[5]

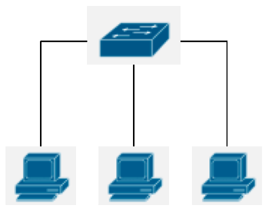

Figure 2: *Basic DHCP Setup With Cisco Switch.* This self-contained test environment consists of a set of computers connected to a single Cisco switch (a DHCP server, the rogue DHCP service, and our prober P). This setup was also used for the forwarding detection scenario.

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 [2]. This is the sort of exploit that motivates us to examine DHCP closely.

Our prototype software uses Scapy scripts to probe 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.

6.3 Environment

We used three main environments for our DHCP experiments. First, as shown in Figure 2, we have a small, self-contained test environment consisting of a set 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.

We also have access to the 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. Finally, we have the DHCP server included in a Linksys WRT54G2 router. It runs as the core of a small home network.

As described previously, 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 another server. 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 compared responses to probes across our test environments. Doing so simulates the introduction of another DHCP agent onto the network whose traffic we are seeing instead of the legitimate server's traffic. This process is akin to comparing a previous behavior model captured in a known trustworthy state with a later behavior model gleaned from an environment during recovery. We can probe the server at a time when we assume it to be in a trustworthy state; this can be established by (1) manual inspection of the program or process, (2) some kind of integrity check of the code and configuration files a la Tripwire, or (3) immediately after a new deployment of the service.

6.4 Constructing DHCP Probes

Within a DHCP packet (see Figure 4), four 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 for the chaddr field:

- The client's MAC address
- Another valid MAC address
- An all-zeroes MAC address
- An all-ones (Broadcast) MAC address

```
# Request probe w/ ciaddr set to other IP
state = random.getstate()
probeFunc(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,
                 xid=random.randint(0, 4294967295))
          /DHCP(options=[("message-type", "discover"),
                         ("end")]
               ), state)

# Request probe w/ chaddr zeroes
state = random.getstate()
probeFunc(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",
                 xid=random.randint(0, 4294967295))
          /DHCP(options=[("message-type", "discover"),
                         ("end")]
               ), state)

# Request probe with chaddr nonsense
state = random.getstate()
probeFunc(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'',
                 xid=random.randint(0, 4294967295))
         /DHCP(options=[(``message-type'', ``discover''),
                        (``end'')]
              ), state)
```

Figure 3: *Example Probes for DHCP*. Three of the eleven probes we constructed for profiling the behavior of a DHCP server. We took a profile of the known good DHCP service and compared it against another profile from a different machine.

We also send discover probes that manipulate option values. These include normal options and the parameter request list, which allows the requesting client to ask for specific information from the server. We set a number of options in our probes and assign them values (where applicable) as described above. We also send a number of probes requesting different information from the server using the parameter request list option (we do not believe that Scapy implements all of the options).

6.5 Results

The tool successfully identified that significant differences exist between the production DHCP server and the Linksys router. Not only were the configurations dif-

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     op (1)    |   htype (1)   |   hlen (1)    |   hops (1)    |
+---------------+---------------+---------------+---------------+
|                            xid (4)                            |
+-------------------------------+-------------------------------+
|           secs (2)            |           flags (2)           |
+-------------------------------+-------------------------------+
|                          ciaddr  (4)                          |
+---------------------------------------------------------------+
|                          yiaddr  (4)                          |
+---------------------------------------------------------------+
|                          siaddr  (4)                          |
+---------------------------------------------------------------+
|                          giaddr  (4)                          |
+---------------------------------------------------------------+
|                                                               |
|                          chaddr  (16)                         |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
|                                                               |
|                          sname   (64)                         |
+---------------------------------------------------------------+
|                                                               |
|                          file    (128)                        |
+---------------------------------------------------------------+
|                                                               |
|                          options (variable)                   |
+---------------------------------------------------------------+
```

Figure 4: *DHCP Message Format*. This diagram was copied verbatim from RFC2131 [13].

ferent, 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.

7 Discussion & Future Work

We discuss how our active probing methodology would apply to two other critical network services (DNS and ARP). This is in essence future work, but we offer the sketches as evidence of the feasibility of extending this type of probing to other fundamental network services. We are currently extending our analysis (and crafting probes) to other services like SNMP, STP, NTP, and routing protocols. Each of these protocols requires a different type of approach to composing a verification plan, since their modes of operation may not naturally fit a query-response pattern. In such scenarios, we can take advantage of the cross-layer data pattern and trust relationship patterns.

In the two examples below, our probing has different semantics than our “rogue DHCP” and “forwarding detection” examples. Since we sketch an outline of a verification plan, we focused on relatively easy ways to verify the trustworthiness of these services (e.g., for DNS comparing against known good answers). We could, however, rely on a behavioral signature much more in line with the DHCP experiment by issuing probes that exercise little-used options or ask for incomplete or illogical DNS and ARP resolutions.

7.1 Domain Name System

We describe one way in which our method might 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 [23, 24]. In many cases, DNS responses are trusted by default, since they represent the best and frequently only information a host has about how to resolve external names to machine addresses. 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 should 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.

In creating a verification plan, we cannot feasibly examine what answers the server gives 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 hosts 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.

7.2 Address Resolution Protocol

We describe how our method could apply to Address Resolution Protocol (ARP). ARP operates on the stimulus-response model where each host or gateway can both make and service requests [28]. 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

Protocol	P	T	X
DNS	Host	DNS Server	Name resolution information
DHCP	Host	DHCP Server	IP address, gateway address, DNS address, etc.
ARP	Host/Gateway	Host/Gateway	IP address:MAC address mapping
STP	Switch	Switch	Bridge priority, path cost
CDP	Network Device	Network Device	Addresses, device information

Table 2: *Enumerating Services Involved in a Network's "Deception Surface"*. The protocols here form the main deception surface of a network; we list them in the context of our data model's trust relationship syntax. Even though there are "secure" variants of some of these protocols, networks do not always use them because requiring authentication infrastructure in order to establish basic layer 2 and 3 connectivity can be cumbersome and difficult to maintain.

active probing.

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.

Merely looking at ARP replies in isolation may not be sufficient. Consider the following scanning strategy. A prober conducts an ARP scan of a given set of addresses, and for each address scanned it does two things. First, it listens for replies and raises an alert if it hears more than one different MAC address in response. Second, if only one response is received, it saves that response to a hashtable. It then would check for one of two conditions. The scan can either look for the address it has just heard appearing in the hashtable twice, or to look for it to not be in the hash.

The prober needs to run this scan against both its local network (excluding the gateway) and against addresses outside its network. The former warns the prober of untrustworthy ARP behavior of hosts and servers on its own segment, and the latter of such behavior associated with its gateway. The scan needs 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 intended for multiple IP addresses going to the same device on the network. This falls outside the definition of trustworthy ARP behavior, and the prober can raise an alert. We could run forwarding detection against the non-gateway IPs which returned the duplicate MAC, but it is not strictly necessary. Probes would take the form of a simple ARP scan, with a supporting hashtable. The technique employs a brute-force approach, but should successfully detect ARP issues on the network.

7.3 Limitations

Although the probing approach we discuss is meant to serve as a kind of "tripwire for trust," it has several shortcomings. Of particular interest going forward is the consideration of how to scale the process of producing a verification plan and the concomitant probes to very large networks (along with large networks containing non-TCP/IP networking equipment). In a sense, the manual nature of writing probing scripts both helps and hinders the ability to scale. On one hand, writing scripts for a small number of critical pieces of network infrastructure benefits from the manual attention to detail and the knowledge of the system administrator about the quirks or peculiarities of the system being probed. On the other hand, in a highly heterogeneous environment containing a network composed over years from a variety of organizations, the sheer diversity of core services poses a significant challenge.

One way to deal with this challenge is to focus on detecting the presence of certain types of deceptions rather than verifying the behavior of every last system. Another (complementary) approach would require research that can attempt to generate a set of probes from pristine (or trusted) configuration files and/or binary code of the target service.

The stimulus-response pattern for detecting untrustworthy behavior may not apply well to protocols that are not purely request-response based (e.g., they may operate on a stream of asynchronous update messages). We can attempt to verify the behavior of such services through trust relationships and cross-layer data (for example, for a routing protocol we might spoof or issue route withdrawals or announcements from one peer and see if the target announces such messages to another peer).

Finally, we have not explicitly considered the effect the use of such active probing might have on IDSs extant in the target environment. It is likely that certain types of IDS might alert on messages from the prober, especially if they are malformed in some fashion. Dangers here include the IDS increasing its alert logging (and thereby increasing the noise in its alert stream or logs) as well as subtly changing their view of the network. In general, a coordinated security response from multiple independent security mechanisms is a hard unsolved problem. Nevertheless, one of our primary use cases is in a network that we are attempting to recover; we might expect to ignore the secondary effects of such probing in favor of re-establishing core critical services.

8 Conclusion

This paper suggests that active intrusion detection (AID) techniques hold promise as a useful network security pattern, particularly when attempting to verify that basic constraints or characteristics of the network hold true. We presented an approach to AID based on probing: issuing crafted packets meant to elicit a particular type of response from the target system or host.

There are several conceptual lessons to take away from this work. Our main approach is predicated on probing the "corner case" behavior and configurations of network services and verifying that services return known–good answers. Our main assumption is that normal behavior is in some sense equivalent to "trustworthy." Feeding a system crafted input meant to exercise corner cases in logic or configuration serves as a good heuristic for revealing behavior that might carry highly individualized information. We hypothesize that meaningful differences in the characteristics of network trust relationships can reveal malicious influence (or at least a bug or misconfiguration).

We suggested three patterns for building verification plans and exploring this space of varied behavior: stimulus-response, cross-layer data, and trust relationships. This approach can help users, client hosts, and system administrators verify the trustworthiness of network services, especially in the absence of strong authentication mechanisms at layer 2 and 3. We discussed how to apply this method to DNS and ARP, we crafted packets that can remotely detect a host in forwarding mode, and we implemented a Scapy-based prototype to verify the trustworthiness of a DHCP service.

9 Acknowledgments

We appreciate the insight and comments of the LISA reviewers. In particular, they asked us to provide more detail on our prototype and data model as well as more carefully discuss the current limitations. Our shepherd, Tim Nelson, showed a lot of patience in working with us to reconcile some of the submission manuscript's shortcomings; the final paper is much improved because of his input and guidance.

Locasto is supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] 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/`.

[2] 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.

[3] ARKIN, O. ICMP Usage in Scanning or Understanding some of the ICMP Protocol's Hazards. Tech. rep., The Sys-Security Group, December 2000.

[4] ARKIN, O., AND YAROCHKIN, F. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. Tech. rep., August 2002. `http://ofirarkin.files.wordpress.com/2008/11/xprobe2.pdf`,.

[5] AXELSSON, S. Intrusion Detection Systems: A Survey and Taxonomy. Tech. rep., Chalmers University of Technology, 2000.

[6] BOWES, R. Scanning for Conficker's peer to peer. *Skull Security* (April 25, 2005). `http://www.skullsecurity.org/blog/2009/scanning-for-confickers-peer-to-peer`.

[7] BOWES, R. Zombie Web servers: are you one? *Skull Security* (September 11, 2009). `http://www.skullsecurity.org/blog/2009/zombie-web-servers-are-you-one`.

[8] BOWES, R. 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`.

[9] CHESWICK, B. An Evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference* (January 1992).

[10] DAVID LAPORTE AND ERIC KOLLMANN. Using DHCP for Passive OS Identification. BlackHat Japan.

[11] DORMANN, W. Vulnerability note vu#154421. *US-Cert Vulnerability Notes Database* (March 5, 2005). `http://www.kb.cert.org/vuls/id/154421`.

[12] DREGER, H., FELDMANN, A., MAI, M., PAXSON, V., AND SOMMER, R. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *Proceedings of the USENIX Security Symposium*.

[13] DROMS, R. Dynamic Host Configuration Protocol, March 1997. `http://www.ietf.org/rfc/rfc2131.txt`.

[14] DUNLAP, G. W., KING, S., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (February 2002).

[15] FRIAS-MARTINEZ, V., SHERRICK, J., D.KEROMYTIS, A., AND STOLFO, S. J. A Network Access Control Mechanism Based on Behavior Profiles. In *Proceedings of the Annual Computer Security Applications Conference* (December 2009).

[16] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Security Symposium* (2001).

[17] HILZINGER, M. Fedora: Chronicle of a Server Break-in. `http://www.linux-magazine.com/linux_magazine_com/online/news/update_fedora_chronicle_of_a_server_break_in`, March 2009. Linux Magazine.

[18] 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`.

[19] LOCASTO, M. E., BURNSIDE, M., AND BETHEA, D. Pushing boulders uphill: the difficulty of network intrusion recovery. In *Proceedings of the 23rd conference on Large installation system administration* (Berkeley, CA, USA, Nov 2009), LISA'09, USENIX Association.

[20] LYON, G. *Remote OS Detection*, nmap reference guide, chapter 8 ed., 2010.

[21] MAREK MAJKOWSKI. sniffer-detect.nse. Nmap Scripting Engine Documentation Portal. http://nmap.org/nsedoc/scripts/sniffer-detect.html.

[22] MARKOFF, J. Worm infects millions of computers worldwide. *The New York Times* (January 22, 2009).

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

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

[25] OZGIT, A., DAYIOGLU, B., ANUK, E., KANBUR, I., ALPTEKIN, O., AND ERMIS, U. Design of a log server for distributed and large-scale server environments.

[26] PETERSEN, B. Intrusion Detection FAQ: What is p0f and what does it do? Tech. rep., The SANS Institute. http://www.sans.org/security-resources/idfaq/p0f.php.

[27] PHILIPPE BIONDI. *Scapy v2.1.1-dev documentation*, April 19, 2010. http://www.secdev.org/projects/scapy/doc/.

[28] PLUMMER, D. C. 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.

[29] PROVOS, N. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 1–14.

[30] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. rep., Secure Networks, Inc., January 1998.

[31] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of* 13^{th} *LISA Conference* (November 1999), pp. 229–238. `http://www.usenix.org/event/lisa99/roesch.html`.

[32] SINGER, A. Tempting Fate. *USENIX login; 30*, 1 (February 2005), 27–30.

[33] SPAFFORD, E. H. The Internet Worm: Crisis and Aftermath. *Communications of the ACM 32*, 6 (June 1989), 678–687.

[34] STOLL, C. Stalking the Wily Hacker. *Communications of the ACM 31*, 5 (May 1988), 484.

[35] 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`.

Notes

[1] `http://www.h-online.com/security/news/item/Worm-uses-built-in-DHCP-server-to-spread-1255388.html`

[2] `http://www.secdev.org/projects/scapy/`

[3] `http://www.rapid7.com/products/nexpose-community-edition.jsp`

[4] `http://www.tenable.com/products/nessus`

[5] Although DHCP in IPv6 operates slightly differently, opportunities for masquerading still exist, and active probing can help detect such attacks.