CS4516: Project Methodology

Tag-Based IP Spoofing Prevention Under Low Deployment Scenarios

Author:
Michael Calder
Daniel Robertson

Supervisor:
Dr. Craig Shue

February 27, 2014

1 Implementation of our extended protocol with NS-3

Our implementation and evaluation is done through the Network Simulator 3 (NS-3) framework. NS-3 is an extensive open source framework for network testing and simulation, written in C++ with Python bindings for higher level scripting. It's seen wide spread use in both commercial and academic pursuits. After learning the NS-3 way of doing things, implementing a novel networking protocol is relatively straight forward. As was the case with this project, simply forking the repository, and extending a few base classes with the desired functionality, allows for rapid development.

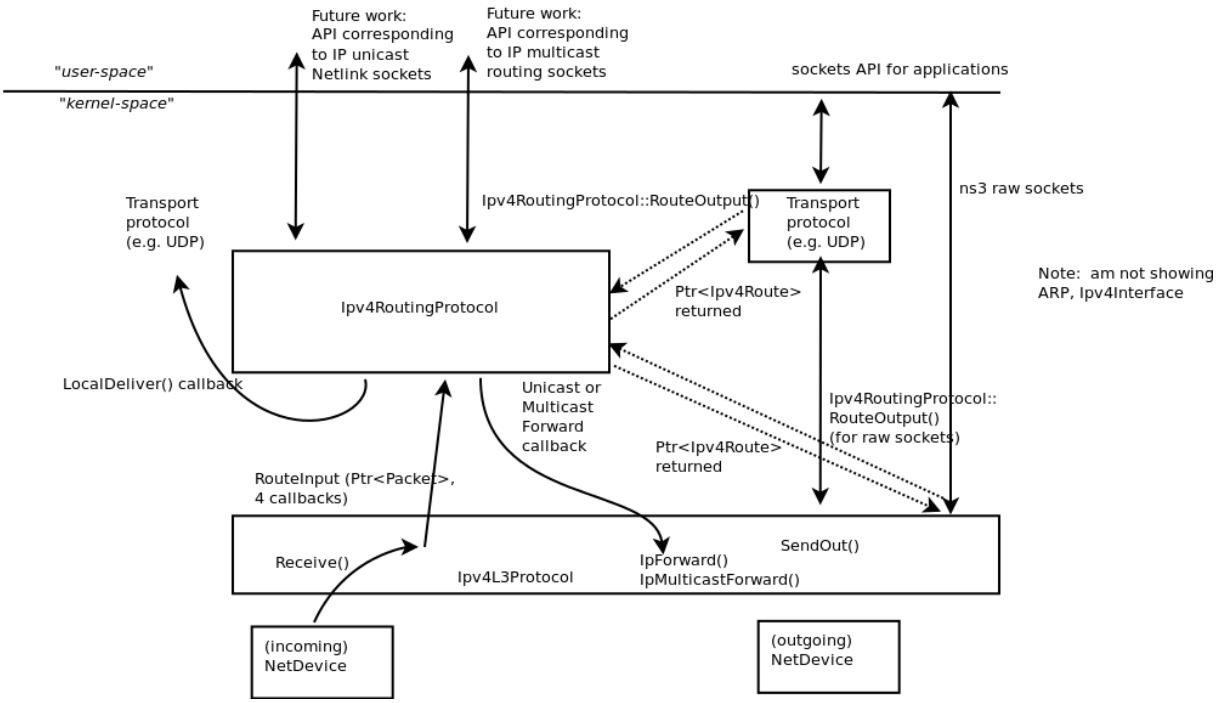

Figure 1: NS-3 `Ipv4RoutingProtocol` diagram. Note the interaction between the simulated layer's through RouteInput and RoutOutput [1]

Since our extended protocol operates on the network layer, we worked mainly with NS-3's `Ipv4RoutingProtocol` interface. We extended the Ipv4GlobalRouting implementation in order to perform the necessary tag protocol logic on a per packet level. The `Ipv4RoutingProtocol` expects a RouteInput and RouteOutput func-

tion to be specified, which handles the control logic of prefix matching and packet forwarding. Here, we created a tag table as specified in [3]. Since our project's focus is more on the exchange of tagged packets over an insecure connection, we assumed that each router's tag table would already be populated. We do this by deterministically calculating a router's tag based off its associated prefix and incoming interface. This allows each router to act as though it's been fully configured, without the added implementation of tag discovery.

Tags are stored within each packets header as a a list of tag objects. Necessary interaction functions (add, get, remove) were added to the `ipv4-header` class to support the addition of a tag field. The RouteInput and RouteOutput hooks both receive a Packet and Header pointer, allowing for this tag data to be accessed and modified by each router in the stream.

These modifications are still insufficient to emulate our extended protocol in an actual deployment simulation. At it's core, an NS-3 simulation manages a group of configurable objects called `Nodes`. A Node can act as an end host, but can also be further extended to include a `NetDevice` (analogous to an NIC) in order to function as a router. `Nodes` are allocated and managed by `NodeContainers`. We use these containers to logically divide multiple `Nodes` into Subnets, each with their own tag. Client Nodes within the subnet are connected over `CSMA` (forming a bus topology). The edge router in each subnet is attached to the main network over a `PointToPoint` link. Each router node is configured before simulation start up by the `GlobalRouteManager` class. We hooked into the configuration process in this class to control the deployment status of our extended protocol on each router. Each node manages an instance of the `GlobalRouter` class (which uses the `Ipv4GlobalRoutingProtocol`), allowing it to function as a router. Since the `Ipv4RoutingProtocol` is designed to work independently of per router information (such as the routers IP address), the `GlobalRouter` class has to communicate this information to the Node's `Ipv4RoutingProtocol` object. This is what allows each router to know and add its subnet's tag to the packet. The extended protocol is disabled by default in our modified `Ipv4GlobalRoutingProtocol` class, so it must be manually enabled via the `GlobalRouter`. When the simulation begins, all Nodes are iterated over, and those with a `GlobalRouter` object aggregated to them are assigned to the protocol depending on the result of a call to `rand` compared against a deployment threshold.

Just as in a normal network, information is encapsulated in `Packets`, wrapped in a number of `Headers`. While in an ideal network, the tags would be implemented in the IP Header, for the sake of convenience, we chose instead to attach the tags to the `Packet` class. The class exposes an API for adding attributes to simulated packets that persist throughout their transmission. Funnily enough, these are simply called `Tags`. Our protocol tags are therefore an extension of NS-3's Packet `Tag` interface.

With a stable modified implementation of the `Ipv4RoutingProtocol` and its constituent classes, we will set up a small topology consisting of multiple different subnets. This novel topology consists of two simulated LANs each with an edge router, attached to each other over two different intermediary routers (each with their own subnets). The ensures that each packet is passes through at least two tagging routers. A UDP echo client/server application is installed on each client Node, allowing back and forth communication over our protocol. Here, we will focus on the generation and verification of secured packet tags, based off of a router's base tag. We devise two core approaches, both of which involve computing the hash of a base tag in order to combine it with the current unix time interval. Both these approaches aim to vary the output tag that is to be transfered over an insecure channel, such that an attacker who obtains it cannot use it to spoof routers inside its originating subnet.

We will be testing both the speed and security of two different approaches to preventing tags from being compromised. For each of these approaches, we will use and compare both the fastest known non-cryptographic hash function (xxHash [2]) and one of the fastest known cryptographic functions (SHA-1).

The first approach is hash-chaining, which involves producing a series of hashes of the original tag to be used each day (changing the seed each day so that no two days have the same hash series). Every hash is the previous digest hashed again. If the time interval that these hashes change each day is 30 seconds, then we will need to produce 2,280 hashes to be stored in the router's RAM each day. We will test how long it takes to brute-force each hash algorithm to decide the maximum time interval that can be used and still render a compromised hash useless after its time interval has ended. We will measure how reasonable the needed memory is for each algorithm and how long it takes for routers to produce each day's chain. We note that for this approach, xxHash is a more attractive option, as it would

allow large chains of small 32 bit hashes to be computed relatively quickly.

The second approach is inspired by TOTP (Time-based One-time Password). We will combine the current time (rounded to some interval like 30 seconds) with the original tag and then hash that result, rendering compromised hashes only useful for a small time interval. Security is a more important factor in this approach because given a time in addition to the hash algorithm, an attacker may be able to obtain the original tag. We will analyze the pre-image resistance of the output using a traditional brute-force approach, in order to determine the resilience of the approach. Clearly, it should be computationally difficult to derive the base tag from a hashed tag, when given the time interval used at the time of hashing, and the hash function its self. This calls for a cryptographic hash function, so we anticipate that SHA-1 and stronger cryptographic functions will be more secure.

We plan to measure the number of hashes that can be computed over a one second time span for both algorithms. In addition, we intend to examine the space complexity associated with the hash-chaining method, particularly the amount of storage required for one day's worth of generated tags. Finally, we will analyze whether tag creation and authentication incurs latency overhead at each router in the topology (to be measured in milliseconds). We will consider our approach successful should this difference is under a millisecond, using either hashing method.

References

[1] Routing overview ns-3 vns-3.10 documentation, February 2014.

[2] xxhash - extremely fast non cryptographic hash, February 2014.

[3] Craig A. Shue, Minaxi Gupta, and Matthew P. Davy. Packet forwarding with source verification. *Computer Networks*, 52(8):1567 – 1582, 2008.