UNIVERSITEIT VAN AMSTERDAM

MSc Security and Network Engineering
Research Project II

Automated end-to-end e-mail component testing

November 20, 2018

Isaac Klop
isaac.klop@os3.nl

Kevin Csuka
kevin.csuka@os3.nl

Supervisor
Michiel Leenaars
NLnet

Assessor
Prof. dr. ir. C.T.A.M. de Laat

Abstract

The first e-mail was sent in the early 70's. In almost 50 years, many components have been added to the e-mail architecture. E-mail software has become complex and now requires a lot of manual configuration, which creates a large surface for human error. In this research, we aim to take away anxiety of managing a mail server by researching how one can prove that a mail server is properly set up via automated end-to-end component testing. We create a taxonomy to identify relevant components in the e-mail architecture. Based on those components we develop an automated, modular and portable test suite. To support the end-to-end tests, we set up seven public mail servers. The mail servers have intentional flaws in configuration and DNS records to simulate different scenarios. In this way, the test suite assures an administrator that IMAP, POP, DKIM, SPF, DMARC, Greylisting, authentication, Sieve, SMTP (MSA) and DANE are set up properly. It partially covers the spam filter and TLS, and it does not cover SRS.

Contents

1 Introduction

Handling electronic mail in the modern age involves many different software components, as well as significant configuration skills and regular maintenance. This creates a large surface for human error. An administrator sets up and configures his mail server. The logs do not show a single error, the Domain Name System (DNS) [1] records are set and are valid. How does he actually know that the mail server is set up properly? How does he know there is not a domain out there rejecting his e-mail? It can be difficult to find the cause of a malfunctioning mail server. This can create anxiety for administrators that wish to manage their own mail server.

What is currently missing is an automated end-to-end e-mail component test that administrators managing mail servers can use to see if all the components in their actual setup are fully functional. Does spam mail get properly classified or rejected? Does the DNS-based Authentication of Named Entities (DANE) TLSA [2] record correspond with the actual certificate, and do the correct ciphers and protocols show up? Do the reports of Domain-based Message Authentication, Reporting and Conformance (DMARC) [3] get sent out?

In this project we attempt to create a portable automated modular end-to-end test that administrators can rely on, and we evaluate its reach.

1.1 Research questions

Based on the introduction, we define the research question as:

- **To what extent can we prove a mail server is properly set up via end-to-end component testing?**

In order to answer the main research question, we define the following sub-questions:

- What are relevant mail server components, and what are their functions?
- How can we verify, through end-to-end testing, if those components work properly?

We expect that most components of a mail server can at least partially be tested. However, the functioning of some elements may only be proven in real use. It may also prove difficult to identify all components and test cases.

2 Related work

nixcloud.email [4] is part of nixcloud-webservices. It focuses on easily deploying and operating a mail server. It deploys Postfix [5], along with several components, on a machine running NixOS [6]. The project includes a simple test suite specifically designed for its implementation. The tests are run when a user deploys or updates his nixcloud.email server. The test suite sets up a temporary virtual mail server to support end-to-end testing. It tests, for example, the sending and receiving of e-mail, aliases, spam filtering and mailbox quota. This approach allows an administrator to test his configurations and software versions for errors.

MxToolbox [7] is an online service providing mail related diagnostics. The tool tests several components such as Transport Layer Security (TLS) support, various DNS records and whether the mail server acts as an open relay. Open relays are frequently used to send spam [8]. However, the toolbox does not test how the mail

server handles these settings. It sets up a connection to the server to verify the settings but it does not send or receive e-mail.

Internet.nl [9] provides an easy-to-use mail server test service. The user submits an e-mail address and receives a test report. The service verifies the syntax and policies of various DNS records, checks for Internet Protocol version 6 (IPv6) support and provides an extensive test for TLS. It checks the cipher suites offered by a mail server against a list published by the Dutch National Cyber Security Centre (NCSC) [10]. Similar to MxToolbox, Internet.nl does not actually send or receive e-mail.

mail-tester [11] tries to see whether a received e-mail is likely to be classified as spam by examining the headers and content. Other than Internet.nl and MxToolbox, mail-tester comes closer to the end-to-end testing that we are interested in, because it can receive e-mail. mail-tester validates Sender Policy Framework (SPF) [12] and Domainkeys Identified Mail (DKIM) [13], it checks the domain against various blacklists and it uses a spam filter.

The website `www.emailsecuritycheck.net` [14] provides a web service to verify the working of a single component, the spam filter. The user provides an e-mail address to which the web service sends seven malicious e-mails. All of them are expected to be rejected or classified as spam.

Microsoft's Remote Connectivity Analyzer [15] is a web-based tool that helps administrators to identify connectivity issues with their mail servers. The tool is designed for Microsoft's own mail server software, MS Exchange, but some tests work on other e-mail software as well. The tool tests whether a remote user can send and receive e-mail from a mail server.

Microsoft's Remote Connectivity Analyzer, mail-tester and emailsecuritycheck.net do end-to-end testing, but they test a small set of components. Internet.nl and MxToolbox are more extensive in their approach but lack end-to-end tests. Also, none of the mentioned tools are automatable, they all require some form of user input to run.

3 Method

3.1 Taxonomy

We identify relevant components in the e-mail architecture. We do this by creating a taxonomy. In the taxonomy, we divide the e-mail architecture into six message agents. Four of the message agents are standardized in RFC 5598 [16], the Internet Mail Architecture. These are the:

- Message Delivery Agent (MDA), e.g. Sieve [17]
- Message Submission Agent (MSA), e.g. Postdrop [18]
- Message Transfer Agent (MTA), e.g. Postfix [5]
- Message User Agent (MUA), e.g. Thunderbird [19]

We identify a fifth message agent, the Message Retrieval Agent (MRA), e.g. Fetchmail [20], and a sixth, the Message Access Agent (MAA), e.g. Dovecot [21]. We consider the functions of an MRA as described on the Mutt wiki [22]: "A Mail Retrieval Agent makes connections to a remote mailbox and fetches mail for local use". The MAA is sometimes referred to as Mail Access Server. RFC 8314 [23] describes it as: "The term "Mail Access Server" refers to a server for POP, IMAP,

and any other protocol used to access or modify received messages, or to access or modify a mail user's account configuration".

For each message agent, we use related work on e-mail testing, as well as online guides for mail servers, to determine which components are part of that message agent.

3.2 End-to-end testing

Paul [24] describes E2E testing as: "E2E testing is similar to integration testing; however, E2E testing focuses exclusively from the end user's point of view while integration testing can focus on any subset of subsystems". In our case, we connect to the mail server in the same way a normal user, via his MUA, would. We create a test suite that tests both the sending and receiving of e-mail. We run our tests from an end-to-end (E2E) perspective. To follow the framework Paul describes, we treat the mail server we are going to test as a black box.

For this research we assume a mail server provides the functions of an MTA, an MSA, an MDA and an MAA. These four agents make up our *test object*. We consider the MUA and MRA client software, and we will ignore these in our tests.

The test suite uses the Simple Mail Transfer Protocol (SMTP) to submit e-mail messages to the MSA of the mail server and use the Internet Message Access Protocol (IMAP) to retrieve messages from the MAA [25, 26].

The benefit of this approach is that the administrator can test the full functionality of the system and that when the tests succeed, he can be confident that the system will work in real world scenarios. By treating the mail server as a black box, we also create a test suite that is platform-agnostic. The downside of this approach is that it is difficult to pinpoint the exact module that is causing the failure since we have no access to the configuration or logs. When a test fails, we point at the component that failed the test, but we, for example, do not determine whether the failure is caused by faulty software or faulty configuration.

We want it to be possible to include the test suite in an automated deployment pipeline to support Continuous Integration and Continuous Delivery (CI/CD). All tests in the test suite should run without input from the user. To support an automatic rollback mechanism, the test suite should return an exit code that the user can check for.

3.3 Proof of Concept

We develop a test suite to answer the main research question. The test suite should be modular, so it should be possible to enable or disable specific tests without affecting the behavior of the other tests. The test suite should be portable as well, it should be usable on most systems.

We choose Python version 3 as the programming language because this is installed by default on many operating systems [27]. We organize the test suite using *unittest* [28], a unit testing framework for Python. The *unittest* framework has a *skip* feature that can be used to enable/disable specific tests. We use *smtplib* [29] to send e-mail and, we use *imaplib* [30] to retrieve e-mail.

The test suite requires an account on the test object that can send and receive e-mail. The credentials for this account are only used by the test suite itself and never leave the machine running the test suite.

3.3.1 End-to-end setup

To achieve our end-to-end design we require one or more mail servers from which e-mails are sent to the test object and from which we can retrieve e-mails sent by the test object. The mail servers must be reachable by both the test object and the test suite. In this section, we identify three possible methods of setting up these mail servers. One where we set up virtualized mail servers on the same host as the test suite. One similar approach where we also virtualize the test object itself. And one where we set the mail servers up on another machine, virtual (Virtual Machine (VM)) or bare metal, and provide them with public IP addresses.

We can not assume that the test suite is run on a publicly reachable machine. In the first approach, the VMs will most likely not have public IP addresses. It will not be possible then, to run our test suite from a machine other than the machine that hosts the test object. A downside of this approach is that setting up all required VMs requires considerably more resources than running just the test object. Running the tests sequentially, setting up and tearing down VMs in between, requires fewer resources, but increases the overhead of the test suite. An upside of this approach is that it contains the entire test process to a single host. The administrator is in control of the virtual machines. He can easily modify settings to his desire. As the virtual machines run locally, access to them is most likely restricted. This makes it easier to secure the test environment.

In the second approach, where we also virtualize the test object. We take the exact software and configuration, and we rebuild a virtual copy along with the other mail servers. As stated in Section 2, Related Work, the nixcloud.email project does something similar in their test suite. An upside of this approach is that it allows testing a different configuration without actually deploying that configuration. This approach also allows the user to run the test suite on any machine, resources are less of a concern. The downside of this approach is that the test suite is not fully run "from the user's point of view". The virtual copy of the test object can not use the same DNS records as the actual test object. In this approach, a series of non end-to-end tests will have to cover the test object's DNS records.

The third approach requires almost no extra resources for the user. However, in this approach, anyone will have access to the mail servers. This creates a burden of managing and securing those servers. The most important downside is that the servers require an authentication mechanism to prevent abuse, some kind of registration process for example. Someone has to manage the servers, update them, create backups and manage the authentication. The upside is that the tests come as close as possible to the normal use of the test object. This approach also allows regular users, other than administrators, to test mail servers for which they have credentials.

We choose to set up the mail servers according to the third approach, hosting them publicly. Because the mail servers will be hosted in a different administrative domain, the test e-mails may pass through intermediary networks, firewalls or intrusion prevention systems. Although this approach creates a security burden, it comes closest to the real world use of the test object. The goal of this research is to take away anxiety. The closer the test suite resembles real world use, the more confidence it can instill.

The mail servers that we set up are named *mail-test-x*, where x is replaced by a number. In the rest of the paper we refer to those mail servers by that name. We refer to the mail server that is being tested by the test suite as *test object*.

3.3.2 Process

As an example, the process of testing whether the test object can send e-mail, as seen from the test suite, is shown in Figure 1. Each number correlates to a step as shown in the list below. The figure shows two mail servers, the test object and a *mail-test-x* server, *mail-test-01* in this case.

1. Connect to the test object using *smtplib* and send an e-mail to a test address on *mail-test-01*
2. The test object sends the e-mail to *mail-test-01*
3. *mail-test-01* submits the e-mail in the INBOX corresponding with the test address
4. Connect to *mail-test-01* using *imaplib* and verify that the e-mail arrived in the INBOX

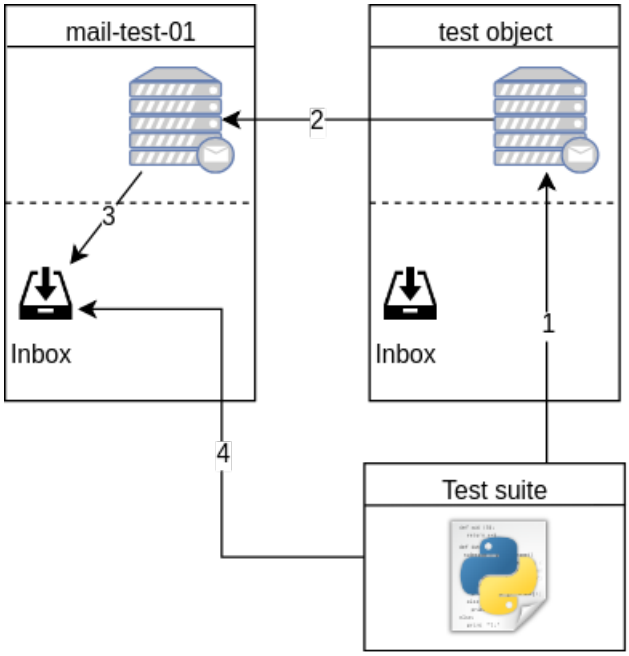

Figure 1: Example of the process of the test suite verifying whether the test object can send e-mail

4 Results

In this section we show the taxonomy of the e-mail architecture, we elaborate on the *mail-test-x* domains, and we elaborate on our test design.

4.1 Taxonomy

We create a taxonomy based on the e-mail architecture. The taxonomy is shown in Figure 2. We use several sources to include relevant components for the taxonomy.

We use the e-mail testing services described in section 2, Related Work. We identify the different components that these services test, which we then include in the taxonomy.

The Dutch Standardisation Forum [31] provides a list of mandatory and recommended open standards for Dutch government and semi-government organizations. The list includes many open standards related to e-mail. We use this list to identify components.

We also use online "how to" guides on setting up mail servers [32, 33, 34, 35]. These guides often set up different components on top of the mail server software, which we can add to our taxonomy.

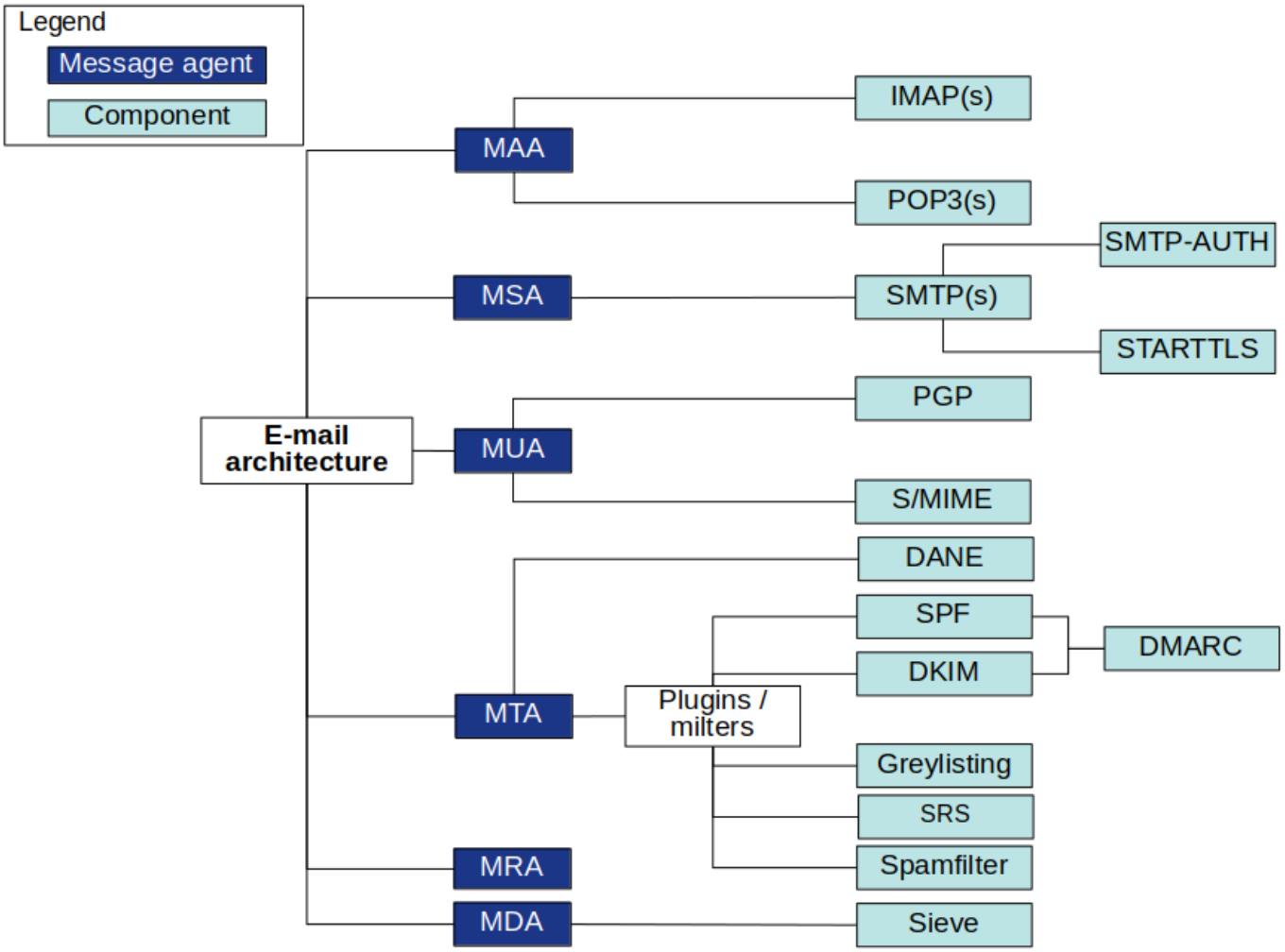

Figure 2: Taxonomy of the e-mail architecture

Each component shown in the taxonomy is explained from Section 4.3.1 to Section 4.3.12.

4.2 Public mail servers

To verify the test object's behavior, we perform end-to-end tests with various parameters. To perform these tests on all the listed components, we require seven additional mail servers (*mail-test-[1-7]*). Each mail server has specific characteristics in terms of configuration of the mail server and its DNS records. There is, for example, one mail server with an SPF record policy of *"v=spf1 -all"* in its domain. Any e-mail sent from that domain has to be rejected. This example is shown in Figure 3. We follow the same process for DKIM, DMARC, DANE etc.

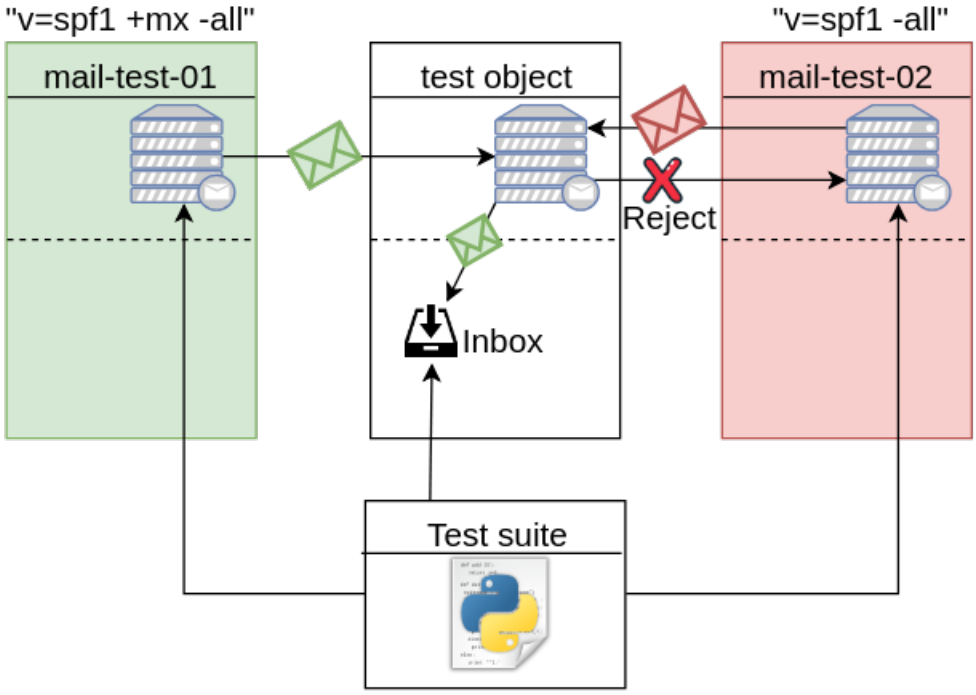

Figure 3: Process of verifying how the test object handles SPF

Currently, a user using our end-to-end test has two options: Use our servers or deploy the servers himself. We automate the installation and configuration of the seven mail servers using Ansible [36]. We add the Ansible playbook, along with our test suite, to a Git repository [37]. For convenience, Ansible lists the required DNS records of the *mail-test-x* domains. The user can add this list to his DNS.

When the test suite uses the seven mail servers that we provide, we have to distribute credentials for those mail servers to the user running the test suite. A lasting solution is outside the scope of this proof of concept, for now we provide those credentials only to trusted users.

Table 1 shows the *mail-test-x* servers and their configuration.

4.3 The Test Suite

The automated end-to-end e-mail component testing tool is available on a Git repository [37]. We developed the tool to use mostly standard Python 3 libraries and it is possible to switch a test on or off without affecting the other tests. The README in the Git repository describes the use of the tool.

Hostname	**Mail configuration**	**DNS configuration**
mail-test-01	SPF DMARC DMARC reporting	Valid SPF record Correct DKIM record DMARC policy none (p=none) Valid TLSA record
mail-test-02	Greylisting	Valid SPF record Correct DKIM record
mail-test-03	Force DANE SPF	Valid SPF record Correct DKIM record DMARC policy none (p=none) Invalid TLSA record
mail-test-04	SPF	Hardfail SPF record (v=spf1 -all) Correct DKIM record DMARC policy reject (p=reject)
mail-test-05	DMARC	Valid SPF record Incorrect DKIM record
mail-test-06	*none*	No SPF record Incorrect DKIM record DMARC policy reject (p=reject)
mail-test-07	No TLS	TLSA record

Table 1: Used *mail-test-x* servers, their mail configuration and DNS records

A component can have one or more tests associated with it. The basis for each test is the same. We send an e-mail to or from the test object, we retrieve the e-mail from the relevant mailbox and we check one or more headers.

Each test has a unique ID, derived from a time stamp and an incrementing counter. This ID is used for the subject of the e-mails. This subject is then used to distinguish the e-mails in the mailbox.

When a test fails, the test suite provides feedback to the user. The feedback shows:

- the name of the test
- the ID of the relevant e-mail
- what the test suite expected to happen
- what actually happened
- the name of the component that is most likely the cause

The user can look up the name of the test in the test suite's documentation [37] for more information on the test.

Table 2 shows which component tests we implemented in the proof of concept. In the following sections we describe the tests and limitations in more detail.

Components	Implemented
IMAP	✓
SMTP (MSA)	✓
POP	✓
Authentication (MSA)	✓
Authentication (MAA)	✓
TLS (MAA)	partial
TLS (MSA)	partial
DANE	✓
SPF	✓
DKIM	✓
DMARC policy	✓
DMARC reporting	✓
SRS	✗
Greylisting	✓
Spam filter	partial
Sieve	✓

Table 2: Components which the test suite can and cannot verify

A complete list of the tests is shown in Appendix B.

4.3.1 SMTP (MSA), POP and IMAP

IMAP and Post Office Protocol (POP) are protocols used to grant users remote access to their mailbox. In our case, where we perform end-to-end tests on the test object, we require IMAP to provide remote access to the received e-mails. In order to send e-mails from the test object, we also require remote access to its MSA.

We create one test for IMAP/POP and one for the MSA. The test is simple, we send an e-mail from the test object to a *mail-test-x* domain and vice versa. We then list the content of the respective *INBOX* with both IMAP and POP to verify that the e-mail arrived.

4.3.2 Authentication (MSA and MAA)

The MSA and MAA require a form of authentication: The MAA to ensure the mailbox cannot be read by other users and the MSA to ensure only authorized users can use the mail server to send e-mail. Without MSA authentication, the mail server becomes an open relay.

We can test this, again, by sending an e-mail from the test object and by listing the *INBOX* of a user. For both actions, we require the test object to ask for authentication first.

Once authenticated to the MSA, the user can send e-mail on behalf of the mail server's domain. We test if the test object correctly verifies whether the authenticated user is authorized to send e-mail on behalf of the user set in the *MAIL FROM* header. For example, if *Alice* successfully authenticates as *alice@example.com* she should not be able to send an e-mail with *bob@example.com* as the *MAIL FROM* header.

4.3.3 TLS (MSA and MAA)

As of RFC 8314 [23], mail servers that have an MSA or an MAA must support TLS for those agents. We test this by sending an e-mail from the test object and by listing the *INBOX* of a user. For both actions, we expect the test object to require a TLS connection.

We do not check the cipher suites offered by the test object. To test this in an end-to-end setup it would require creating one *mail-test-x* server per different combination of TLS version and cipher suite, meaning the amount of required servers for a single test increases significantly. A unit test, such as the one performed by Internet.nl [9], provides the same assurance as such an elaborate end-to-end test would. On their website, Internet.nl states that they will soon publish their source code [38]. When this happens, we can update the test suite to include this test.

4.3.4 DANE

RFC 7672 [39] describes DANE for SMTP as: "This specification uses the presence of DANE TLSA records to securely signal TLS support and to publish the means by which SMTP clients can successfully authenticate legitimate SMTP servers". DANE TLS is resistant to downgrade attacks and man-in-the-middle attacks [39]. RFC 7672 elaborates more on DANE for e-mail and SMTP, and RFC 7671 [40] is the latest RFC on DANE itself. DANE relies on DNS Security Extensions (DNSSEC) [41]. To support DANE, a sending mail server must have a local DNSSEC validating recursive resolver.

The DANE component covers the TLS for MTA to MTA communication. RFC 7672 describes opportunistic DANE: "With opportunistic DANE TLS, traffic from SMTP clients to domains that publish "usable" DANE TLSA records in accordance with this memo is authenticated and encrypted. Traffic from legacy clients or to domains that do not publish TLSA records will continue to be sent in the same manner as before, via manually configured security, (pre-DANE) opportunistic TLS, or just clear text SMTP."

In the TLS for MSA and MAA tests, we verify whether the test object supports TLS for the MSA and MAA. The tests described in the next paragraphs verify, by verifying DANE, whether the test object supports TLS for MTA to MTA communication. As with the TLS for MSA and MAA tests, we do not verify the cipher suites offered by the test object.

Receiving A sending mail server may validate DANE when setting up an SMTP session with another mail server. To allow this, the test object should have a valid TLSA record in the DNS. There are three scenarios to test for: the test object can have no TLSA record, a valid TLSA record or an invalid TLSA record. An invalid TLSA record is one where the key or certificate does not match the key/certificate served by the mail server.

We send two e-mails to the test object, one from a mail server that uses opportunistic DANE and one from a mail server that forces DANE. We expect both e-mails to arrive to verify if the test object has a valid TLSA record.

The mail server that forces DANE terminates the connection before an e-mail is sent when there is no TLSA record. This scenario is shown in Figure 4. In the figure, *mail-test-04* forces DANE. *mail-test-03* uses opportunistic DANE and will in the scenario where the test object does not have a TLSA record, still deliver the e-mail.

If neither e-mail arrives, we conclude that the test object has an invalid TLSA record. If only the e-mail from *mail-test-03* arrives, the test object has no TLSA record. If both e-mails arrive, the test object has a valid TLSA record.

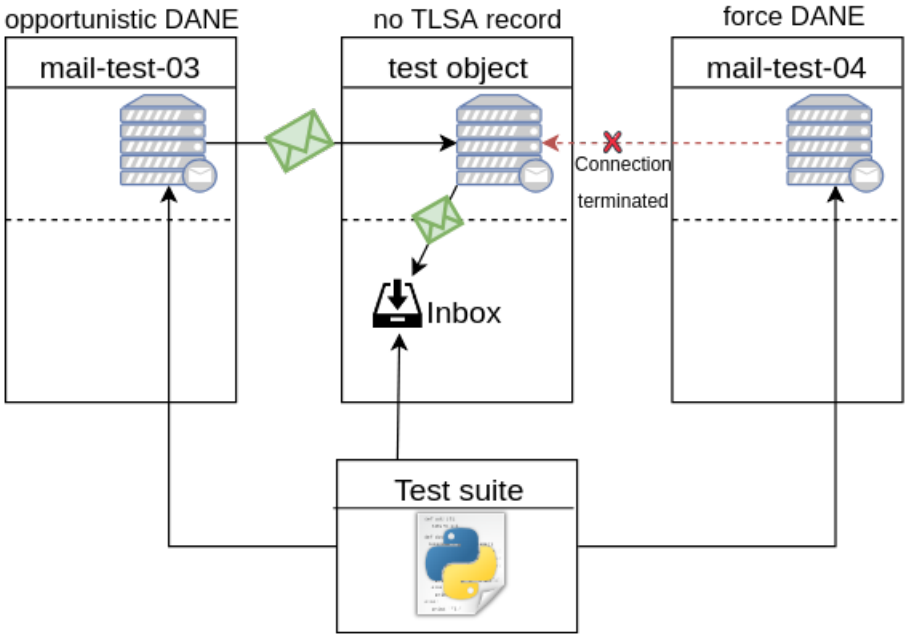

Figure 4: Verifying the TLSA record on the test object when test object receives e-mail. In this example, the test object has no TLSA record. Connection is terminated if DANE is forced

Sending A sending mail server using opportunistic DANE should be able to send e-mail to a mail server with a valid TLSA record as well as a mail server without a TLSA record. It should not be possible to send to a mail server with an invalid TLSA record. To protect against downgrade attacks, it should also not be possible to send an e-mail to a server with a TLSA record when that server does not offer TLS. We test for all four scenarios: how the test object handles sending e-mail to a mail server with a valid TLSA record, an invalid TLSA record, a TLSA record but no TLS and without a TLSA record.

4.3.5 SPF

SPF aims to verify whether a sender is authorized to send on behalf of the domain he advertises in the *HELO* and in the *MAIL FROM* header [12]. The sender places a TXT record in the DNS that specifies which mail servers are authorized to send e-mail on behalf of the domain.

Receiving The SPF component in the test object that validates SPF may return one of seven result values as specified in RFC 7208 [12]. These are: None, Neutral, Pass, Fail, Softfail, Temperror and Permerror. The RFC also provides guidance on how to respond to each of these values. It should be noted that RFC 7208 describes: "there is no comprehensive normative requirement for message handling

in response to any particular result". The guidance is just that, there are no strict requirements for handling the return values. In our test, however, we are trying to prove whether a mail server is properly set up. Our definition of proper in this case corresponds to the *Result Handling* section of the RFC. We feel that administrators that deviate from the guidance of the RFC do so consciously and are, as such, not alarmed when they fail a test in the test suite.

The administrator can configure the SPF component to either handle validation on its own, the SPF component instructs the MTA to reject or accept an e-mail based on its validation. Or, the SPF component can add an additional header field that can later be used by another component to determine whether to accept the e-mail.

In our test, we test whether the test object accepts an e-mail when the result should be *None*, *Neutral* or *Pass* and whether the test object rejects the e-mail when the result should be *Fail*. We also extract the header from a received e-mail and verify that the result value in the header is what it should be according to our test case.

We can not test whether the test object handles a *Softfail*, *Temperror* or *Permerror* result correctly. The RFC does not provide guidance on how to handle such results. It leaves the result handling, for those results, up to the administrator. For *Softfail*, the RFC does specify that the mail server *SHOULD NOT* reject an e-mail based solely on a *Softfail* result, but we can not determine whether our test object rejected an e-mail solely on the basis of the SPF validation.

Sending Sending an e-mail that can be verified with SPF requires an SPF record in the DNS. We test two cases, one where we send an e-mail from the test object to a *mail-test-x* domain with a valid SPF record. And one where we spoof the *MAIL FROM* header, so the SPF validation should fail. We then check how our receiving SPF component validates the e-mail.

In the first test case, we check the SPF authentication header to determine whether the test object has a valid SPF record that authorizes it to send e-mail on behalf of its domain. In the second case, the SPF authentication header should show that the *mail-test-x* server, is not authorized to send on behalf of the test object's domain.

4.3.6 DKIM

DKIM uses a public/private key pair to sign e-mails. The public key is placed in the DNS of the sender, in the form of a TXT record. The sender signs the e-mail with a digital signature in the e-mail header. The receiver can verify this signature using the public key set in the DNS record [13].

Receiving As described in RFC 6376 [13], the evaluation of the DKIM signature can have three states; SUCCESS, PERMFAIL and TEMPFAIL. However, the action to be taken for each state should have no effect on handling the receiving e-mail. RFC 6376 describes this as: "Survivability of signatures after transit is not guaranteed, and signatures can fail to verify through no fault of the Signer. Therefore, a Verifier SHOULD NOT treat a message that has one or more bad signatures and no good signatures differently from a message with no signature at all".

We expect the DKIM component at the test object to communicate the result of the DKIM verification by adding a verification header.

We create a *mail-test-x* domain which signs e-mails with the correct key. We also create a *mail-test-x* domain which signs e-mail with a different key than what it publishes in the DNS. We send an e-mail from both servers to the test object and, we determine whether the test object adds the correct verification headers. This process is shown in Figure 5.

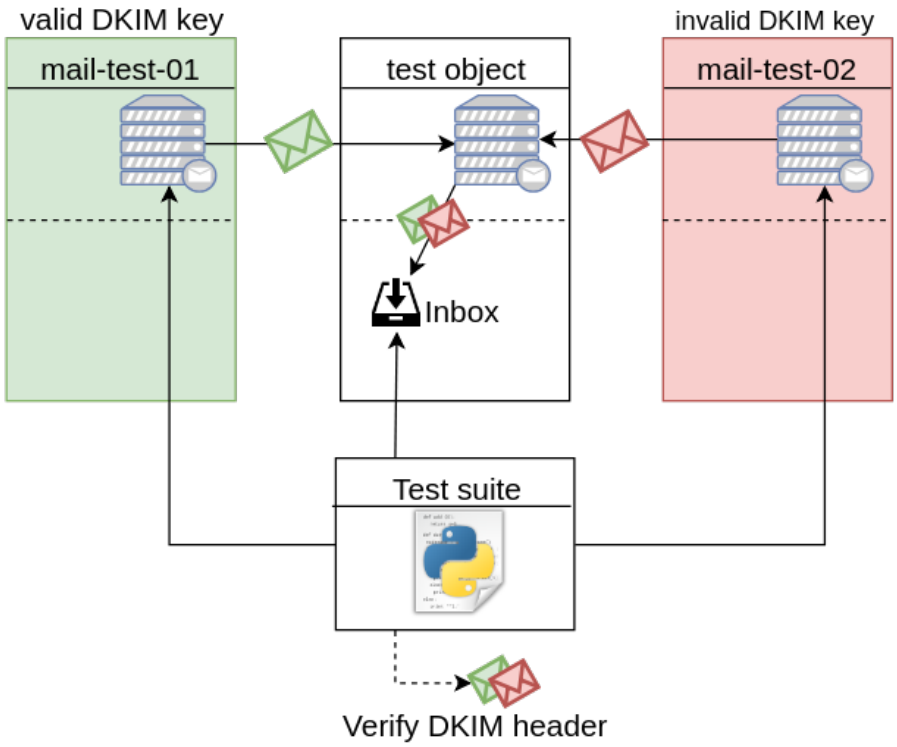

Figure 5: Verifying how the test object handles DKIM signatures

Sending We verify that the test object correctly signs its outgoing e-mail. In this test, we also count the number of signatures in the outgoing e-mail. An administrator may have, unknowingly, set up multiple components that do DKIM signing. While the specification allows this, we still warn the user when the test finds multiple signatures.

The test fails when an outgoing e-mail has one or more signatures that fail to validate or when the outgoing e-mail has no signatures at all.

4.3.7 DMARC - Policy

DMARC is a reporting, authentication and policy protocol combining SPF and DKIM. RFC 7489 [42] describes DMARC as: "DMARC allows domain owners and receivers to collaborate by sending assertions of the domain owners' policies to the receiver and provide feedback to the sender for monitoring authentication".

Receiving The owner of a domain publishes a DMARC DNS record. Upon receiving an e-mail from such a domain, the DMARC component checks both SPF

and DKIM, and applies a certain policy when one or both of the checks fail. The policy is set in the DNS record at the sender's domain, which can be either None, Reject or Quarantine.

We verify whether the test object applies the correct policy when it receives an e-mail from a domain that publishes a DMARC record. We do this by configuring a *mail-test-x* domain with a bad DKIM key and a DMARC policy of *reject*. The DKIM verification at the test object should return a fail and the DMARC component should apply the *reject* policy specified in the DNS record. We verify that the e-mail is indeed rejected.

Sending We expect the test object to have a DMARC record with a sufficient policy, either *reject* or *quarantine*. A policy of *none* would allow senders with a spoofed address to send e-mails to the domain.

We verify the test object's policy by sending a spoofed e-mail to a *mail-test-x* domain. We spoof the e-mail by setting the *MAIL FROM* header to the e-mail address of the test account, while we send the actual e-mail from another *mail-test-x* domain. We then expect the e-mail to be either rejected or quarantined.

4.3.8 DMARC - Reporting

DMARC also provides a reporting scheme where domain owners send reports to each other to provide feedback and to report failures [42].

Receiving The e-mail address to send these reports to is set in the *rua* (feedback) and *ruf* (failures) field of the DMARC DNS record. For this test to work, the test suite requires the credentials of the account where the DMARC reports are sent to. Our *mail-test-x* server sends out two DMARC reports every minute, one to the *rua* address and one to the *ruf* address. We verify that the test object receives the DMARC reports.

Sending We expect the test object to send out DMARC reports of its own. The interval between reports can be specified in the DMARC record of the domain receiving the reports. A domain sending out DMARC reports must be able to send a report once every 24 hours and should be able to send a report every hour [42]. A one-hour interval is also the minimum interval, the receiver of the report may specify a value lower than that but a reporting MTA is not forced to honor that [43].

The test verifies whether the test object sends out a DMARC report to a *mail-test-x* domain within 24 hours of receiving an e-mail from that domain. The test will wait for up to 24 hours, checking for DMARC reports at one-hour intervals, before failing. RFC 7489 [42] describes that: "anything other than a daily report is understood to be accommodated on a best-effort basis." To account for the best-effort basis, the test expects a report within 24 hours. It will, however, report to the user if it receives reports before that. The DMARC record at the *mail-test-x* domain specifies a requested reporting interval of one hour.

The test is optional. Since we have to account for the use case where our test suite is used in a CI/CD pipeline, we do not always want to wait this long before returning the test results.

We test whether the test object sends out a DMARC report. We do not verify the content of the report. Section 7 of RFC 7489 provides a list of data that a

DMARC report should contain. It is possible to use this list to verify the content of a DMARC report, but this requires automatically processing the e-mail attachment. We have no control over the content of an attachment, the content can be malicious. Automatically processing the content is a complex and potentially dangerous operation. An operation that the test suite would carry out on the user's machine.

4.3.9 SRS

If a domain forwards a message from another domain, the Return-path is changed to the address of the account that forwards it. This may result in delivery issues when the receiver validates SPF. The e-mail might be rejected or marked as spam. This mechanism is shown in Figure 6.

Sender Rewrite Scheme (SRS) is a method to fix this. It is a way for forwarding MTAs to rewrite the envelope sender MAIL FROM header, so it passes the SPF check at the receiver [44].

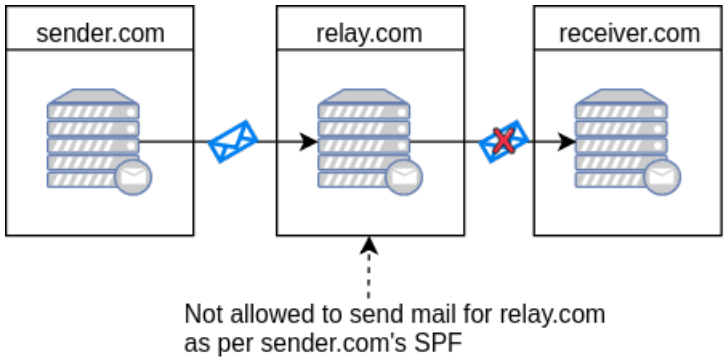

Figure 6: SPF blocking e-mail forwarding

When the test object acts as a mail relay for another domain, it should apply SRS. To verify if the SRS is applied correctly, we require credentials of the mail server for which the test object acts as a relay. Because we treat the test object as a black box, we can not control for which domains the test object acts as a relay. We can not configure one of our *mail-test-x* domains as a relay domain on the test object.

If the test object acts as a relay for an existing domain, we would require credentials of that domain. We decide to exclude this test from the test suite because, for security implications, we do not want to store the credentials of another domain.

4.3.10 Greylisting

Greylisting is an anti-abuse system that temporarily rejects e-mail from unknown sources. RFC 6647 [45] describes greylisting as: "Broadly, the term refers to any degradation of service for an unknown or suspect source, over a period of time (typically measured in minutes or a small number of hours). The narrow use of the term refers to generation of an SMTP temporary failure reply code for traffic from such sources."

RFC 6647 recommends a mail server to: "Include a configurable range of time within which a retry from a greylisted host is considered and outside of which it is otherwise ignored." The RFC also specifies that: "The default range SHOULD be from one minute to 24 hours. Retries within the range are permitted and satisfy the greylisting test, and the client is thus no longer likely to be a sender of spam. Retries after the end of the range SHOULD be considered to be a new message for the purposes of greylisting evaluation."

Sending When the test object sends an e-mail to a *mail-test-x* server that does greylisting, the *mail-test-x* server first responds with an SMTP temporary failure reply code.

We test this by sending an e-mail from the test object to a *mail-test-x* server that does greylisting. This e-mail should be rejected at first. After a timeout of one minute, as configured in *mail-test-x*, the e-mail should be accepted whenever the test object tries to re-send the e-mail. For the test, we expect the test object to re-send the e-mail within 10 minutes of being first rejected.

Receiving We expect the test object to adhere to the recommended time range. We send an e-mail from a *mail-test-x* domain to the test object. The test object should temporarily reject this e-mail. We verify that the e-mail is indeed rejected and after a time out of one minute, we re-send the e-mail. We then verify that the e-mail arrived at the test object.

This test covers the low end of the time range. We test the high end of the time range by waiting for 24 hours after the last test. We expect the test object to start greylisting the *mail-test-x* domain again after 24 hours have passed. This test is optional. Users that do not want to wait 24 hours before returning the test results can disable the test and test the high end of the time range by simply running the test suite again after 24 hours have passed.

4.3.11 Spam filter

There are several spam filter software implementations available to reject unsolicited e-mails [46]. Each spam filter has their own default configuration and some spam filters are even self-learning [47]. An administrator can configure the spam filter to his own liking. It is therefore difficult to create a predictable test for the spam filter.

SpamAssassin created a Generic Test for Unsolicited Bulk Email (GTUBE) [48]. The GTUBE is a specific string that can be set in the message body. Spam filters are not forced to recognize this test string but most implementations do.

We test the spam filter on the test object by sending an e-mail with the GTUBE string to the test object. We verify that the e-mail is either rejected or marked as spam.

4.3.12 Sieve

Sieve [17] is a language for e-mail filtering at time of final delivery. It is designed to be used not only by administrators, but also regular users.

ManageSieve [49] is an addition to Sieve, it is a protocol that allows users to manage their Sieve scripts remotely. The test uses ManageSieve to submit a Sieve script to the test object. We verify whether the test object correctly applies the

rules in the Sieve script by sending an e-mail to the test object that matches a rule in the script.

5 Conclusion

The goal of our research was to take away the anxiety of administrators by creating a comprehensive automated test suite that assures mail server components work properly via end-to-end testing.

We identified relevant e-mail components and placed those in a taxonomy. Based on the taxonomy, we developed a test suite consisting of 31 tests. A complete list of these tests is shown in Appendix B.

Table 2 shows to what extent we can prove that a mail server is properly set up. The test suite assures an administrator that IMAP, POP, SMTP (MSA), Authentication, TLS, DANE, SPF, DKIM, DMARC, greylisting and Sieve are properly set up. We are limited in proving that TLS cipher suites and the spam filter are properly set up, and we do not prove whether SRS is properly set up.

We verify whether the mail server supports TLS, but we do not verify whether the correct cipher suites show up.

We verify whether the mail server has a spam filter that supports SpamAssassin's GTUBE pattern. We cannot verify the spam filter if it does not support the GTUBE, and we do not verify whether the spam filter correctly identifies spam beyond the GTUBE.

The tool tests a large part of the mail server components in the e-mail architecture. This should take away some of the anxiety around managing a mail server. The tool is open source and may form the basis for a comprehensive automated test suite that tests all aspects of a mail server.

6 Discussion

The test suite proves that certain components or parts of components are set up correctly. Our research depends on the taxonomy we created. The taxonomy is based on other work on mail servers and e-mail (security) testing. There are many components in the e-mail architecture and even more different software implementations of those components, many of them glued together with various glue software such as milters, proxies and plugins. The taxonomy covers all standardized e-mail components, but for certain functions, e.g. blacklisting, spam and malware filtering, there are implementations that are not covered in the taxonomy.

Our test suite is designed to work on all mail server software. The functionality of a mail server, from an end-to-end perspective, should always be the same. However, we have not tested our test suite on all mail server software.

There are many software implementations for each component. Ideally, we would like to test the test suite on not just different MTAs, but also on different component software and the various combinations one can make with these components. We have tested the test suite on a mail server running *Postfix*, *OpenDKIM*, *OpenDMARC*, *Rspamd*, *pypolicyd-spf* and *Dovecot* [50, 51, 47, 52, 21]. This covers a subset of the many combinations of component software implementations. There may be software that we have not covered that may cause false positives or false negatives.

This research only covers end-to-end testing. Unit or integration tests may provide a user with more granular feedback. However, unit or integration tests may not always test the full functionality of a component. Some tests, such as those that verify whether a DNS record is set and valid, may benefit from the granularity of unit or integration testing.

Although the modular test suite allows turning on/off tests, we believe that an administrator has to set up all tested components correctly in order to have a fully functional mail server. We expect the administrator to pass all required tests. In creating the test suite, we have to make decisions on when to mark a test as "failed" or when to warn the user. We are, inevitably, opinionated. Since we are testing a black box, we can not use the configuration of the test object to determine when a test should fail. For each component, we consult the respective RFC or specification on how to handle different results. The guidance in an RFC often uses the keyword *SHOULD* or *SHOULD NOT* to describe an item. This keyword means that the item can be ignored or deviated from in particular circumstances [53]. Therefore, behaviour of the test object may, in exceptional cases, be different from what our test suite expects.

The user of our test suite has to either use our seven hosted servers or host his own. We want everyone to be able to use our test suite without limitation, but at the same time we have to prevent abuse. This requires some form of registration and authentication before a user can use our servers. When a user hosts his own servers, he shifts the burden of securing and managing them to himself.

7 Future work

The amount of components in the e-mail architecture has been expanding since the first e-mail was sent. The test suite tests are based on the components in the taxonomy. It would be very useful to extend the taxonomy to include all the various software implementations of the e-mail infrastructure components as well as new components. One of the latest relevant drafts is Authenticated Received Chain (ARC) [54]. Once the taxonomy is expanded, one can come back to this research and expand the test suite.

The goal of this research was not to compare existing solutions with our solution, but rather fill the gap. It can be interesting to perform a comparison study of our test suite against existing, perhaps commercial, mail server testing solutions.

There is currently no system to verify whether a user is authorized to run the test suite on his domain. We send around ten to fifteen e-mails to that domain, and we set up as many IMAP connections. In the current setup, we only allow trusted users access to our seven *mail-test-x* domains. In the future, when others want to use the test suite, we require some form of authentication and registration mechanism.

There are many techniques for classifying an e-mail as spam. There has been research into comparing and evaluating these techniques, such as Blanzieri and Bryl [55]. One can use this and similar research to create a test suite for spam filters.

8 Acknowledgements

We would like to thank Michiel Leenaars from NLnet for his active support and supervision during this project.

References

[1] P. Mockapetris. Domain names - implementation and specification. STD 13, RFC Editor, November 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[2] P. Hoffman and J. Schlyter. The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa. RFC 6698, RFC Editor, August 2012. http://www.rfc-editor.org/rfc/rfc6698.txt.

[3] M. Kucherawy and E. Zwicky. Domain-based message authentication, reporting, and conformance (dmarc). RFC 7489, RFC Editor, March 2015. http://www.rfc-editor.org/rfc/rfc7489.txt.

[4] nixcloud. nixcloud.email. https://github.com/nixcloud/nixcloud-webservices/blob/master/documentation/nixcloud.email.md. Accessed, Oct. 19 2018.

[5] Wietse Venema. The Postfix Home Page. http://www.postfix.org/start.html, 2018. Accessed, Nov. 02 2018.

[6] NixOS. About NixOS. https://nixos.org/, 2018. Accessed Nov. 19 2018.

[7] MxToolbox, Inc. MxToolbox. https://mxtoolbox.com/SuperTool.aspx. Accessed, Oct. 10 2018.

[8] IBM. Open relays. https://www.ibm.com/support/knowledgecenter/zh/SSKTMJ_8.5.3/com.ibm.help.domino.admin85.doc/H_OPEN_RELAYS_DETAILS.html, 2009. Accessed, Oct. 15 2018.

[9] Dutch Internet Standards Platform. About the email test. https://en.internet.nl/test-mail/. Accessed, Oct. 10 2018.

[10] NCSC. ICT-beveiligingsrichtlijnen voor Transport Layer Security (TLS). https://www.ncsc.nl/actueel/whitepapers/ict-beveiligingsrichtlijnen-voor-transport-layer-security-tls.html, November 2014. Accessed, Oct. 19 2018.

[11] MailPoet & AcyMailing. Test the Spammyness of your Emails. https://www.mail-tester.com/. Accessed, Oct. 10 2018.

[12] S. Kitterman. Sender policy framework (spf) for authorizing use of domains in email, version 1. RFC 7208, RFC Editor, April 2014. http://www.rfc-editor.org/rfc/rfc7208.txt.

[13] D. Crocker, T. Hansen, and M. Kucherawy. Domainkeys identified mail (dkim) signatures. STD 76, RFC Editor, September 2011. http://www.rfc-editor.org/rfc/rfc6376.txt.

[14] Byteplant Software Solutions. Free Email Security Check. https://www.emailsecuritycheck.net/index.html. Accessed, Oct. 15 2018.

[15] Microsoft. Remote Connectivity Analyzer. https://testconnectivity.microsoft.com, 2018. Accessed, Oct. 17 2018.

[16] D. Crocker. Internet mail architecture. RFC 5598, RFC Editor, July 2009. http://www.rfc-editor.org/rfc/rfc5598.txt.

[17] P. Guenther and T. Showalter. Sieve: An email filtering language. RFC 5228, RFC Editor, January 2008. http://www.rfc-editor.org/rfc/rfc5228.txt.

[18] Wietse Venema. postdrop - Postfix mail posting utility. http://www.postfix.org/postdrop.1.html, 2018. Accessed, Nov. 15 2018.

[19] Mozilla. Software that makes e-mailing easier. https://www.thunderbird.net, 2018. Accessed, Nov. 15 2018.

[20] Matthias Andree. Fetchmail - the mail-retrieval daemon. https://sourceforge.net/projects/fetchmail/, 2018. Accessed, Nov. 15 2018.

[21] Timo Sirainen. Dovecot, Secure IMAP server. https://dovecot.org/, 2018. Accessed, Nov. 02 2018.

[22] Mutt Project. Mutt Documentation - Mailconcept. https://gitlab.com/muttmua/mutt/wikis/MailConcept, 2018. Accessed, Oct. 15 2018.

[23] K. Moore and C. Newman. Cleartext considered obsolete: Use of transport layer security (tls) for email submission and access. RFC 8314, RFC Editor, January 2018. http://www.rfc-editor.org/rfc/rfc8314.txt.

[24] Raymond Paul. End-to-end integration testing. In *Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on*, pages 211–220. IEEE, 2001.

[25] J. Klensin. Simple mail transfer protocol. RFC 2821, RFC Editor, April 2001. http://www.rfc-editor.org/rfc/rfc2821.txt.

[26] M. Crispin. Internet message access protocol - version 4rev1. RFC 3501, RFC Editor, March 2003. http://www.rfc-editor.org/rfc/rfc3501.txt.

[27] Python. Python. https://www.python.org/, 2018. Accessed, Oct. 08 2018.

[28] Python. unittest - Unit testing framework. https://docs.python.org/3/library/unittest.html. Accessed Oct. 09 2018.

[29] Python. smtplib - SMTP protocol client. https://docs.python.org/3/library/smtplib.html. Accessed Oct. 08 2018.

[30] Python. imaplib - IMAP4 protocol client. https://docs.python.org/3/library/imaplib.html. Accessed Oct. 08 2018.

[31] Forum Standaardisatie. Lijst verplichte open standaarden. Online, Forum Standaardisatie, September 2018. https://www.forumstandaardisatie.nl/sites/bfs/files/Lijst_verplichte_open_standaarden_sept-2018.pdf.

[32] European Commission. Guidelines and best practices for the Postfix email service (MECSA). https://mecsa.jrc.ec.europa.eu/en/postfix, 2018. Accessed, Oct. 19 2018.

[33] Cullum Smith. How To Run Your Own Mail Server. https://www.c0ffee.net/blog/mail-server-guide/, 2017. Accessed, Oct. 19 2018.

[34] Skelleton. Build Your Own Email Server on Ubuntu: Basic Postfix Setup. https://www.linuxbabe.com/mail-server/setup-basic-postfix-mail-sever-ubuntu-14-04, 2015. Accessed, Oct. 19 2018.

[35] Xiao Guo-An. How to eliminate spam and protect your name with DMARC. https://www.skelleton.net/2015/03/21/how-to-eliminate-spam-and-protect-your-name-with-dmarc/, 2018. Accessed, Oct. 19 2018.

[36] Red Hat, Inc. Ansible is Simple IT Automation. https://www.ansible.com/. Accessed, Oct. 12 2018.

[37] Isaac Klop, Kevin Csuka. https://github.com/csuka/automated_end2end_email_component_testing, 2018. RP2 - End-to-end automated email component testing, Accessed, Nov. 20 2018.

[38] Internet.nl. Copyright. https://en.internet.nl/copyright/, 2018. Accessed Nov. 19 2018.

[39] V. Dukhovni and W. Hardaker. Smtp security via opportunistic dns-based authentication of named entities (dane) transport layer security (tls). RFC 7672, RFC Editor, October 2015. http://www.rfc-editor.org/rfc/rfc7672.txt.

[40] V. Dukhovni and W. Hardaker. The dns-based authentication of named entities (dane) protocol: Updates and operational guidance. RFC 7671, RFC Editor, October 2015. http://www.rfc-editor.org/rfc/rfc7671.txt.

[41] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Dns security introduction and requirements. RFC 4033, RFC Editor, March 2005. http://www.rfc-editor.org/rfc/rfc4033.txt.

[42] M. Kucherawy and E. Zwicky. Domain-based message authentication, reporting, and conformance (dmarc). RFC 7489, RFC Editor, March 2015. http://www.rfc-editor.org/rfc/rfc7489.txt.

[43] dmarc.org. FAQ - Wiki. https://dmarc.org/wiki/FAQ, 2016. Accessed, Oct. 15 2018.

[44] Julian Mehnle, James Couzens. Sender Policy Framework, SRS: Sender Rewriting Scheme. http://www.openspf.org/SRS, 2018. Accessed, Oct. 19 2018.

[45] M. Kucherawy and D. Crocker. Email greylisting: An applicability statement for smtp. RFC 6647, RFC Editor, June 2012. http://www.rfc-editor.org/rfc/rfc6647.txt.

[46] Apache Software Foundation. Apache SpamAssassin. https://spamassassin.apache.org/, 2018. Accessed, Oct. 20 2018.

[47] Rpamd LTD. Rspamd spam filtering system. https://rspamd.com, 2018. Accessed, Oct. 20 2018.

[48] Apache Software Foundation. https://spamassassin.apache.org/gtube/. The GTUBE, Accessed, Oct. 10 2018.

[49] A. Melnikov and T. Martin. A protocol for remotely managing sieve scripts. RFC 5804, RFC Editor, July 2010. http://www.rfc-editor.org/rfc/rfc5804.txt.

[50] The Trusted Domain Project. OpenDKIM. http://opendkim.org/, 2018. Accessed, Nov. 02 2018.

[51] The Trusted Domain Project. OpenDMARC. http://www.trusteddomain.org/opendmarc/, 2012. Accessed, Nov. 02 2018.

[52] Scott Kitterman. pypolicyd-spf. https://launchpad.net/pypolicyd-spf, 2017. Accessed, Nov. 02 2018.

[53] Scott Bradner. Key words for use in rfcs to indicate requirement levels. BCP 14, RFC Editor, March 1997. http://www.rfc-editor.org/rfc/rfc2119.txt.

[54] K. Andersen and B. Long, Ed. and S. Blank, Ed. and M. Kucherawy, Ed. Authenticated Received Chain (ARC) Protocol. Internet-Draft draft-ietf-dmarc-arc-protocol-18, IETF Secretariat, October 2018.

[55] Enrico Blanzieri and Anton Bryl. A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review*, 29(1):63–92, 2008.

A Acronyms

ARC Authenticated Received Chain. 19

CI/CD Continuous Integration and Continuous Delivery. 4

DANE DNS-based Authentication of Named Entities. 2

DKIM Domainkeys Identified Mail. 3

DMARC Domain-based Message Authentication, Reporting and Conformance. 2

DNS Domain Name System. 2

DNSSEC DNS Security Extensions. 11

E2E end-to-end. 4

GTUBE Generic Test for Unsolicited Bulk Email. 17

IMAP Internet Message Access Protocol. 4

IPv6 Internet Protocol version 6. 3

MAA Message Access Agent. 3

MDA Message Delivery Agent. 3

MRA Message Retrieval Agent. 3

MSA Message Submission Agent. 3

MTA Message Transfer Agent. 3

MUA Message User Agent. 3

NCSC National Cyber Security Centre. 3

POP Post Office Protocol. 10

SMTP Simple Mail Transfer Protocol. 4

SPF Sender Policy Framework. 3

SRS Sender Rewrite Scheme. 16

TLS Transport Layer Security. 2

VM Virtual Machine. 5

B Test suite component tests

Table 3 shows which tests the suite contains to test the components of a mail server.

Send/receive	Component	Method	Expected result
Send	Greylisting lower boundary	Send mail from all mail servers	Mail should be rejected for 1 minute, because we expect greylisting. Then the mails should arrive, unblocking us from the greylisting, so we can continue tests
Send	First send mail	Send a mail and verify SMTP	Mail should be sent
Receive	First receive mail	Receive a mail and verify IMAP	Mail should be received
Send	TLS availability	Verify STARTTLS is available on target	TLS is available on SMTP
Receive	TLS availability	Test if TLS is available for IMAP	TLS is available on IMAP
Send	TLS forcing	Test if TLS is forced for SMTP	TLS is forced on SMTP
Receive	TLS forcing	Test if TLS is forced for IMAP	TLS is forced on IMAP
Receive	POP3	Test if target can receive POP, by sending mail from mail-01 to target	Mail arrives

Receive	POP3 TLS	Test if TLS is available for POP, by connecting via pop	TLS is available
Receive	POP3 TLS forcing	Test if TLS is forced for POP	TLS is forced
Receive	SPF hardfail	Send mail from mail-04 to mail server, mail-04 has "v=spf1 -all" record	Mail should be rejected
Send	SPF	send a mail pretending to be the mail server to mail-04 which checks SPF	Mail should be rejected
Receive	DKIM bad signature check	Send mail from mail-05, which has a bad DKIM signature	Mail in inbox or junk, with `dkim=fail` or 'DKIM_INVALID' string in header
Receive	DKIM good signature check	send mail from mail-01, which has a good DKIM signature	mail received in inbox, with `dkim=pass` or 'DKIM_VALID' string in header
Send	DKIM signature	Send mail from mail server to mail-01	`dkim=pass` string for all signatures in the header
Send	Sieve	1. Login via managesieve 2. Place and activate a test script which places mail from mail-01 in Junk folder 3. Send mail from mail-01 and verify 4. Place an empty script and delete previous test script	Mail should be in Junk
Receive	Spamfilter	Send mail from mail-01 to mail server with GTUBE string	Mail should not be in the inbox
Send	Greylisting	Send a mail to mail-02 (mail-02 does greylisting). This test waits at most 10 minutes before failing	After rejecting, the mail is re-sent and should be in the inbox of mail-02
Send	Send as other user	Send mail from mail server pretending to be postermaster@domain while authenticated as the test user	Mail should not be sent
Send	Open relay	Test if test object allows the sending of e-mail before authentication	Is not allowed to send when not authenticated

Receive	OpenDMARC report rua	Test if test object receives a ruf DMARC report sent from mail-01, mail-01 sends out a DMARC report every minute	Report is received
Receive	OpenDMARC report rua	Test if test object receives a rua DMARC report sent from mail-01, mail-01 sends out a DMARC report every minute	Report is received
Receive	DMARC policy	Does test object handle mail correctly when a DMARC record specifies Reject? mail-06 is configured with a bad DKIM key and a DMARC policy of reject	Mail should be rejected
Send	DMARC policy	Send a mail pretending to be from the test object to mail-05, mail-05 only uses OpenDMARC to evaluate a sender	Mail should be rejected
Send	DMARC reporting	Test if test object correctly sends out DMARC reports	Receive a DMARC report from test object
Receive	DANE TLSA record	Test if test object can receive email from a domain that validates DANE	Mail should be received
Receive	DANE no TLSA record	Can test object receive mail from a domain that expects a valid TLSA record, test fails if there is no or a bad TLSA record	Mail should be received
Send	DANE validator	Send mail to mail-03, mail-03 has a bad DANE record	Test object should not set up a connection, so mail should not arrive
Send	DANE no TLS	Send mail to mail-07, mail-07 has a TLSA record but does not offer TLS	Test object should not set up a connection, so mail should not arrive
Send	DANE strictness	Send mail to mail-04, mail-04 has no TLSA record	Mail should arrive
Receive	Greylisting upper boundary	Does test object correctly start greylisting again after 24 hours	Test object should greylist after a 24 hour timeout

Table 3: List of tests in the test suite