An insight into the life cycle of testing
critical security updates supporting
large scale security infrastructure

Vittalkumar Mirajkar
Vittalkumar_Mirajkar@mcafee.com

Narayan Naik
Narayan_Naik@mcafee.com

Rakesh Mallinapalli
Rakesh_Mallinapalli@mcafee.com

Abstract

Software security infrastructure requires daily critical updates to help protect against emerging security challenges. Albeit, releasing a timely critical HotFix[HF] update is extremely critical, thorough testing prior to the release, forms an integral part of the release cycle to avoid false positives. This high confidence testing demands a test framework that is machine-driven and can continuously test and release security updates within a limited time window.

The test framework to certify the update should be capable of executing tests on 100’s of machines, with varying versions of critical software, supporting all possible OS versions. The entire exercise of initiating machine-driven testing to automated result compilation and analysis needs to complete within a stipulated time frame for release readiness.

Keeping such a mission-critical, high availability environment operational and triggering test for validation followed by a quick turnaround of the large result analysis to certify release readiness, is a continuous complex firefighting exercise.

Through this paper, we present a case study that demonstrates a framework, tools and techniques that support the demanding nature of rapid security patch testing. This test framework enables quick, consistent, and reliable methods to meet an ever-changing threat landscape.

Biography

Vittalkumar Mirajkar is a Software Architect at McAfee, with 15+ years of testing experience ranging from device driver testing, application testing and server testing. He specializes in testing security products. His area of interest is performance testing, soak testing, data analysis and exploratory testing.

Narayan Naik is a Software Engineer at McAfee, with 11+ years of experience in exploratory testing and performance testing. He holds an expertise in providing consultation to enterprise customers for features and compatibility of various security products and security solutions deployed. His areas of interest are inter-compatibility test areas, performance testing and encryption product lines.

Rakesh Mallinapalli is a Software Engineer at McAfee with 5+ years of experience in building full stack automation frameworks. He is a passionate developer who has used ML and Data analysis to build highly efficient CI/CD test beds.
1. Introduction

Maintaining computer security of large-scale infrastructure demands a daily assessment and patching for latest vulnerabilities through HF deployment. Timely availability of the critical patches plays a major role in this. HFs are released in extremely tight schedules. This requires a much shorter testing time. Supporting timely testing and release of HF patches across a wide spectrum of critical software products is a complex moving puzzle. You need a best-in-class Continuous Integration/Continuous Delivery (CI/CD) framework to do so which can support daily test and release cycles.

2. What we want to test

Our goal is to test and certify each critical HF patch for daily release readiness. The release readiness is to be completed from build availability to release certification within a tight time schedule of a couple of hours.

These could be a single patch or multiple patches that need to be tested on multiple product versions installed on a spread of supported OS's. To understand the complexity in the number of products, operating systems, and environments we are required to certify against, consider the following:

Product set $P_n = \{P_1, P_2, \ldots, P_{12}\}$

Each Product further has “m” versions which are active in the field, there by the product version($V_m$) set is $V_m = \{V_1, V_2, \ldots, V_{10}\}$

We could restrict the per product version active in production to 10

$P_{nV_m} = \{P_{1V1}, P_{1V2}, P_{2Vm}, P_{2V1}, P_{2V2}, \ldots, P_{10V1}, P_{10V2}, \ldots, P_{10Vm}\}$

Each element in $P_{nV_m}$ is to be tested on supported OS's.

$O_{Sn} = \{OS_1, OS_2, OS_3, \ldots, OS_n\}$

If we consider the various stages an OS is present from its release till EOL, each OS itself has multiple flavors and multiple patch versions active in the field.

Example:

Windows 10 has had 12 releases (Windows_10_Versions), each version having 4 flavors supported (Microsoft, Windows 10 editions 2021) i.e Windows 10 Home, Pro, Pro for Workstations, Business.

Considering just Windows released versions till date, Windows have had 15 major releases (Microsoft, Operating System Version 2020)
In our case this was

\[ Pn = 1 \text{ to } 12 \]
\[ PnVm = 1 \text{ to } 10 \]
\[ OSn = 1 \text{ to } 15 \]

Taking 10 major versions per OSn release, have 4 flavors each

\[ OSnVm = \text{OS Version released} \]
\[ OSnVmFx = \text{Flavors active per version} \]
\[ OSnVm, \text{ where } m \text{ range is } 1 \text{ to } 10 \text{ and } OSnVmFx, \text{ where } x \text{ range is } 1 \text{ to } 4 \]

Total number of combinations, theoretically possible is

\[ Pn*Vm*OSn*Vm*Fx = 12*10*15*10*4 = 72000 \]

**Note:**

Not all OS’s releases have same number of released field versions (examples: OS1 may have had 2 releases, whereas OS2 may have had 5 releases.)

Not all Products have same version of releases (example: P1 had 1 to 10 patch releases, where P2 had 1 to 8)

### 2.1 Data sanitization:

As the 1st step creating a workable set, we need to identify which data set is relevant. Visualizing the Product vs OS supported as a 2-dimension table helps us.
Rule for sanitizing the relevant data:

1. Not all product versions are supported by all OS’s.
2. Latest version of product series / versions is not supported by earlier version of OS’s. This information helps in optimizing the product vs OS supported versions.
3. Installation count of specific older version on older OS’s could be insignificant and help us ignore it if the total installation count is insignificant.

2.2 Relevance of data:

Following channels helps us pick relevant data

1. Product Installation telemetry
2. Product install counts from Product Team / Product Managers
3. OS Market share (Statcounter 2021)
3 Building the test bed:

We need Continuous Integration / Continuous Delivery test setup which meets the following specific requirements

1. High availability
2. Has 2\textsuperscript{nd} level redundancy setup
3. A status monitor dashboard
4. Automated Build release to test readiness infrastructure

3.1 Building beyond 2(N+1) Level Redundancy

We borrowed concepts of data center power supply 2(N+1) (Data Center Redundancy: 2N, N+1, 2(N+1) Explained 2019) redundancy and build a test infrastructure to go beyond 2(N+1) redundancy to maintain the availability of test machines.

Mirrored test infrastructure is setup in two different geos to achieve 2N redundancy. The mirror sync is triggered every 12 hours to maintain same operational state.

In each of the setup, each operating system image state can be reverted to a known good state if the test needs to be retriggered. This helps us achieve within a given setup N+N redundancy. In case the test image is corrupted, the monitoring dashboard has a 2\textsuperscript{nd} level redundancy failsafe which kicks in to trigger machines from the 2\textsuperscript{nd} backup rig for testing. This makes it a 2(N+N+N) redundancy setup.

Once the relevant data is available, we apply 80-20 (Armand Ruiz Gabernet 2017) of data analysis to define a product vs OS relevance for test prioritization.

The prioritization is defined as

\textit{OS deployment relevance } \times \textit{Product installation relevance}.

Once this rule is applied, the new optimized test set appears similar to data presented in the table.
3.2 The Test execution controller:

We need a one stop controller which helps the user understand the current state of test execution as well as to auto trigger any remediation required, making an engineer’s manual intervention the last resort. The controller dashboard needs to meet the following requirements.

- Front end which visualizes an easy-to-understand current flow of execution.
- Calls out current run state and failure (if any) and steps taken to remediate it.

The test controller needs to be backed by an intelligent backend which understands current state and kicks in remediation as and when required. One of the important steps in this chain is Power on Self-Test (POST)

POST is done to check test setup readiness to trigger validation cycles.

This includes (but is not limited to):

<table>
<thead>
<tr>
<th>Failure Points</th>
<th>Remediation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network connection</td>
<td>Network card reset followed by reboot</td>
</tr>
<tr>
<td>CPU spikes</td>
<td>Poll back followed by reboot</td>
</tr>
<tr>
<td>Hard disk availability</td>
<td>Release Hard disk using VMware api’s</td>
</tr>
</tbody>
</table>
Test Controller is the next important part of this setup. Test controller is built on an Active-Passive model (Raghumb.gitbooks.io 2021) to ensure high availability.

3.3 The Hardware

We have used Dell Blade server (Dell PowerEdge 2021) with 16 CPU rig for processing power and for storage we have used Dell-EMC XtremeIO (XTREMIO_X2 2020) two brick solution.

VMware vCenter and vSphere is used as the backbone infrastructure.
4 The execution flow

Patch build availability to release testing follows the below flow for automated test to release readiness cycle.

![Execution flow for Patch test to release cycle](image)

At a deep dive level following are the step wise process.

**Step 1: Heartbeat System**

![Heartbeat System Diagram](image)
Prior to checking for build availability for testing, working state of the test environment is confirmed. Test controller triggers “Heartbeat Service” which checks for test environment machine’s health check. This checks VM node cluster is in working condition (ex: Availability, reachability, and disk availability). This information is stored in “HeartBeat db” for future reference and an email notification is sent to stake holder about the base state of the test environment. Visually, this data is also present in Test Controller dashboard as well. The system has the intelligence built in to check environment integrity, take remediation action

Once the test environment stability is confirmed, build copy system is triggered.

**Step 2: Build copy system**

Test controller periodically does a https long polling to check for build availability in GIT repository. As soon as the build is available, it is copied to local controller repository and next step of test execution is triggered.

**Step 3: Run Test**
Controller triggers tests on the machine set hosted in VM infrastructure for sequence of tests that needs to be validated to certify the build. The results are stored in “Result db” and simultaneously test status is updated in dashboard front end. An email notification is triggered to the test owners. The test run controller has intelligence to take remediation action and intuitively trigger re-runs in case of test failures.

**Step 4: Rollback system**

Once the test runs are completed and results are availability either to block the release or pass the build, the results are archived, and the environment is scrubbed to be ready for next test cycle within 30 minutes of last test completion.

Rollback service handles infrastructure services using VM api’s and does following jobs,

- Image roll back to known good state
- Snapshot (in case an issue detected in previous test run) and roll back
- Replace the images (if corrupted by previous test runs)
- Retained bad nodes for future investigation and fresh copies made from gold images for next run

For one end to end cycle, it is a 6hrs round trip time including POST and environment scrubbing posttest completion.

**5 Test Results**

Each stage has individual results published as email report, stored in reference db and also updated in live status dashboard.

**Heartbeat service:**

Heartbeat service publishes results of the test bed health check and remediation steps taken if any. This gives a summary final countdown system check for next run.
Mid run status from dashboard:

Below is the screen shot from dashboard when test is still in progress. A real time view gives test run status, not just over all test status but each individual module test status as well. This helps us isolate failure modules when a system reports test run as failed.

<table>
<thead>
<tr>
<th>MOVE VMs</th>
<th>UPD</th>
<th>ODS</th>
<th>OFFICE</th>
<th>STATUS</th>
<th>RBS</th>
<th>Disk_Space(MB)</th>
<th>Warnings/Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>20H2X64-MOVE49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>20H2X32-MOVE491</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10RS264-MOVE461</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>3.115</td>
<td></td>
</tr>
<tr>
<td>W81P64-MOVE46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>4118</td>
<td></td>
</tr>
<tr>
<td>10R4X64-MOVE47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>12620</td>
<td></td>
</tr>
<tr>
<td>2K16-MOVE47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>3724</td>
<td></td>
</tr>
<tr>
<td>W81P64-MOVE47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>11134</td>
<td></td>
</tr>
<tr>
<td>2012R2ST-MOVE45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>15415</td>
<td></td>
</tr>
<tr>
<td>10RS55X64-MOVE48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>44494</td>
<td></td>
</tr>
<tr>
<td>2K19S64-MOVE481</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10RS588-MOVE481</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Test completion:

Below is the screen shot when the test run is complete for one of the products, among multiple products which is being tested with this system.

<table>
<thead>
<tr>
<th>MAC Machines</th>
<th>UPD</th>
<th>ODS</th>
<th>OAS</th>
<th>STATUS</th>
<th>RBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC_OS_X_MOJAVE_CORP2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_BIGSUR_CORP-MEDIUMDAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_HIGHSIERRA_CORP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_CATALINA_CORP-MEDIUMDAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_CATALINACONS_NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_MOJAVE-MEDIUMDAT-MEDIUMDAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_MOJAVE_CONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC_OS_X_BIGSUR_CONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The test framework in current state is catering to 6 products. Each products test bed consists anywhere between 8 OS’s (as shown above) to 200 OS’s. The complete test run turnaround time is of 6 hrs.
6 Environment upkeeping challenges:

The major two buckets for environment upkeeping are revisiting relevant product versions for testing, and patching of OS

6.1 Relevant Product versions for testing:

Products released to production themselves go through an increase/decrease in deployment numbers. This needs to be reflected in the testbed used for testing. A quarterly review on the deployment numbers sourced from telemetry, product deployment numbers and Product Program Managers help us scale up or scale down a product test set.

6.2 Patching OS's:

Test bed needs regular upkeeping to maintain sync with Microsoft’s Patch Tuesday’s (Patch Tuesday 2005). This is required to reflect the changing end customer environment due to OS patching/upgrades. OS’s patching is also carried out in an automated fashion with one rig going in for an upgrade while the 2nd back up rig being used for main line testing. This ensures zero downtown during system upgrade cycles.

7 Take always

Rapid test-to-release cycles for automated testing of critical security updates for various combinations of products and OS can be achieved. Some of the areas where this model of testing can be used are,

- For quicker certification of HFs for release readiness.
- For releasing minor fixes quickly to field rather than waiting for monthly release cycles.
- Incremental improvement fixes can be tested and released to production as and when feature is ready and not wait for major release cycles.

This approach helps us go one step closer to achieving a true CI/CD flow for this testing.
References


