Enhancing product security design through product performance testing, a new dimension in testing

Vittalkumar Mirajkar Vittalkumar_Mirajkar@mcafee.com

Sneha Mirajkar SMirajka@cisco.com

Narayan Naik Narayan_Naik@mcafee.com

Abstract

Performance testing is an indispensable part of testing life cycle. In endpoint software's, it is one of the key requirements to be met. Traditional performance testing scope has been around measuring response time, latency and resource availability for high frequency features, e.g. system start-up time, system shutdown time, in web services its request and response (turnaround) times.

Performance optimization has always been concentrated around the high frequency code paths. The logging associated with these paths are also optimized to great degree to increase performance. When an error condition occurs, i.e. a deviation from a known positive path, the optimized code flow diverts to forensic code flow and as much information of the current system state is captured.

Test designers rarely consider the most vulnerable area of performance testing – that of less optimized internal error condition reporting and logging code. These vulnerabilities can be identified and tested using targeted resource utilization tests which force application code to switch from a high frequency code path to a forensic code path. These tests are specifically designed to identify performance flaws that make the system vulnerable to a low-rate denial-of-service(DoS) attack. Low-rate DoS attacks are particularly concerning because they require less attack resources to be deployed against an application to achieve the desired effect.

In this paper, we discuss how we design specific resource utilization tests, which help us expose lesser optimized code areas which could lead to potential DoS attack entry points. We have effectively used this technique to explore how better designed test scenarios for resource utilization, can aid in uncovering performance bottle necks, security vulnerability and DoS attack entry points. The end results of this testing help not only fixing performance issues but also making design changes, making the overall product robust to attacks. This method of testing is applicable to any software. Test case efficiency of these tests are designed not only to help uncover performance issues but also to identify potential vulnerabilities.

Biography

Vittalkumar Mirajkar is a Software Architect at McAfee, with 12+ 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

Sneha Mirajkar is a Software Engineer at Cisco, with 10+ years of experience in software testing and extensive hands-on in test automation using PYTHON, Selenium, PERL, QTP, VBscript, web services testing and functional testing. She has expertise in cloud testing (SAAS) and IAAS, AWS applications.

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.

1 Introduction

Performance design considerations for a software product is as important as the product design itself. What do we test to ensure the product meets all the unforeseen performance challenges it may face in the field? During the product design phase, the emphasis is centered on meeting larger performance guidelines and less emphasis is given to functional deviation paths. These overlooked performance considerations come back to haunt us by making the system performance sluggish and in some cases, leading up to a DoS attacks resulting in complete failure of the product or system being over whelmed by it (Changwang Zhang 2012).

2 Performance Testing – What We Know

2.1 For Desktops and On-Premise Servers

Desktop applications and on-premise server applications are covered by the same general performance criteria guidelines, just at a different scale.

For Windows systems, Microsoft has “The Windows Assessment and Deployment Kit (Windows ADK)” (Microsoft 2018). It’s a collection of tools which helps you to customize Windows images for large-scale deployment. It also aids to test the quality and performance of the system under test.

ADK assessments are tasks that simulate user activity and examine the state of the system. Assessments produce metrics for various aspects of the system and provide recommendations for making improvements. ADK has a standard set of assessments.

- ✓ Boot performance (full boot and fast startup)
- ✓ Hibernate and standby performance
- ✓ Internet Explorer startup performance and security software impact
- ✓ File handling and memory footprint
- ✓ Windows Store Apps

Software vendors add their software to the system’s initial state and run ADK assessment test to see how the system is holding, this however does not exercise the software under test(SUT) but records the effects of its presence in system during ADK assessments. Parts of the SUT does get executed, however they are not the target of specific test execution. The larger goal is to record the effects of SUT’s presence in the system when nothing unexpected is happening. Drawback of this test includes the following:

- Writing custom ADK assessments requires the tester to have a working knowledge of ADK.
- Default assessments rarely expose functionality issues of SUT. There is no targeted functional test executed, no corner case looked at. These tests are good for baseline testing but lack some key components for more robust performance testing.

Example:

- These tests do not cover functional test areas or any error conditions the software would be subjected to or targeted towards.
- The SUT is tested with no-load condition, not even an optimal load condition is considered.

2.2 Performance Testing of Cloud Based Applications

Cloud computing is changing the way applications are deployed, monitored and used by the end users. Cloud offers a virtually infinite pool of resources for computing, storage and networking resources where applications can be scaled as desired. So how do you measure the performance of these applications in the cloud?

Performance is a key factor in testing a web application as it directly impacts the end user experience. Performance testing in cloud is different from that of traditional on-premise hosted applications. Traditionally, the main aim of performance testing is to measure the various parameters such as system throughput, latency with changing number of parallel users accessing your application with different load profiles and various other performance metrics.

Some of the cloud metrics which are important are storage, processing, bandwidth and number of users accessing it at any given point. Scalability, availability, fault tolerance and reliability are the other factors which define a robust cloud infrastructure.

Below are the types of performance tests commonly performed on cloud applications:

- ✓ Stress, load and performance test
- ✓ Browser performance test
- ✓ Latency test and targeting infrastructure test
- ✓ Failover and capacity test
- ✓ Soak test

These tests help to characterize the quality aspects of the application like reliability, security and latency. Some of the areas overlooked in cloud performance are: DB latency, latency between two services, I/O delay, load on individual services, resource availability of each services.

Example:

- When a get request received by DB server in cloud that triggers the maximum number of permitted reads, there arises a situation of DB read request capacity being throttled. This leads to performance dip owing DB latency. This situation is not considered when the performance suite is designed as only turnaround time matters most.

3 Code Execution Frequency

When a software application is designed, it is expected to follow specific standards and attain certain performance expected characteristics. This ensures that all user flows are designed with same amount of scrutiny and critical flows are further combed through for fine tuning. Even though there are open source and commercial tools available for static and dynamic analysis of code (Softwaretestinghelp 2018) (Anderson 2008), they help achieve 100% code test coverage, review coverage and to a great degree, help uncover issues but shed little light on the frequency of the flow execution under real conditions.

We conducted a survey questionnaire to help us understand experienced software developer’s perspective on code frequency when they write code. The survey (Survey by authors 2018) helps us confirm our understanding of the SUT from a black box testing perspective. The developer’s responses helped us segment the SUT into areas where we can begin to target our testing. The segments are defined as shown in the table below

Code Section	% of code lines in overall code base	% of frequency of execution w.r.t entire code base	Examples
High Frequency actions	~20%-30% of code base	~70-80+ % of process time	• Core functionality blocks • DB executing a lookup request • Antivirus monitoring Disk I/O • Firewall inspecting every packet traversing IN/OUT of the machine
Mid Frequency actions	~20%-25% of code base	~10-15+ % of process time	• Lesser called functions • Configuration loads • Garbage collector / teardown code / sequencing / schedulers
Low Frequency actions	~45%-50% of code base	~5+% of process time	• Initialization Routines • Error condition handling • Service start, shutdown, restarts • Content loading or DB initialization
Rare code paths	~5% -10% of code base	~1%-2+% of process time	• Service crash and recovery. • Unexpected exit handling

A simplistic every day example to understand the frequency analogy is that of a ***motorcycle***:

- High Frequency Use - wheels which spin all the time, the engine, fuel injection system etc. Parts which need to run all the time.
- Mid Frequency Use - gear shift, breaks being applied, handle correction for direction.
- Low Frequency Use – use of the rearview mirror, ignition ON/OFF, turn indicators being used.
- Rare Frequency Use - crash guards.

4 Logging

Incorporating logging in code flow is an important, integral part of any commercial software. Logging information helps developers not only to debug issues better, but also helps quality analyst and service/support engineers to do first level of investigation in case of escalations. Logging also helps immensely in code maintenance. It is rare that a commercially released application will not have any form of logging.

The following represent a broad categorization of logging typically incorporated into all sections of code (Yuan, Park and Zhou 2012).

- Fatal
- Error
- Warning
- Information
- Debug

Each logging category has different verbosity levels (Yuan, Park and Zhou 2012) associated with them. The detail of less critical events (i.e. debug) will include events from higher category levels, i.e. debug captures everything which is captured by fatal, error, warning and information. Depending on the software module (e.g. Kernel Mode, User Mode or Web application); different default logging levels are enabled. In most open source software, on an average a line of log code exists for every 30 lines of code.

In a production system, logs are sometimes the only source of problem identification and troubleshooting of an event. Studies have shown that the use of log messages for troubleshooting on average, speed up root cause analysis by 2.2 times (D. Yuan 2011). This motivates developers to add more logging information as it gives a flow of thread-start, code flow and the unfolding of events as they happen. Logging is also crucial to system administrators as they are first responders and assist end users when a system malfunctions.

What to log? How much to log? How little to log? (D. Yuan 2011), there has been no formal study around this and it depends on experience of the developer.

4.1 Developer Bias in Code Modules and Logging

Logging is an integral part of software programming practice as it helps to save important runtime information by adding statements in source code as shown below:

```
Log (level, “logging message %s”, variable);
```

A logging statement typically consists of a logging function and its parameters, including text messages, variables, and a verbosity level.

When we examine a typical code path for any functionality, you may observe something like the simplified snippet below:

```
if (testExpression1)
{
  // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
  // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression3)
{
  // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
}
.
.
else
{
  // statements to be executed if all test expressions are false
}
```

The positive or expected outcomes usually log minimal information to convey the event was a success and for rightful reason, this is expected for performance optimizations. It is when this expected condition fails, and an alternate option path is executed that additional information needs to be captured. To better understand the unfolding events, we tend to focus on capturing current system state information before we attempt the next step. The application moves from performance optimized code to a non-optimized forensic flow, i.e., logging. This switch is implicit, necessary and built into the design of the software.

However, the developer then has a choice of how much or little to log (D. Yuan 2011), invariably when software defects are uncovered in any state of the lifecycle, developers will often enhance logs to capture a better understanding of system state at the time of failure. This approach reduces the diagnosis time for future system anomalies. There are several logging libraries, such as syslog (C. Lonvick 2001) and log4j (The Apache Software Foundation 1999-2018) that provide better logging interface (e.g., multiple verbosity levels). However, in the end, it is the developers’ decisions on when, what to collect and log.

4.2 Area of Interest

For the purposes of this paper, the high frequency code paths are of interest, specifically, the isolation of a single code flow or a sub flow, which under normal conditions would be executed most number of times.

Our focus is to record its behavior and inherent design characteristics to build targeted test cases that force the code from the optimized state to deviate to the forensic (log) state.

Following are some of the guidelines we use to identify high frequency code execution flows and record the events that unfold when execution is in positive flow, as well as when a deviation flow is executed.

- Can the software be broken down in to individual modules, individual functions/code flows, sub flows?
- Are these sub-flows part of multiple flows? A sub-flow reused in multiple flows is an indicator for high frequency code. How easy is it to simulate a high frequency execution flow with minimal effort without complex dependencies?
- Is there an event type that needs to be recorded against a positive flow and a deviation flow? What is the data recorded? How different is it in both conditions?
- Are the events status recorded locally? Updated on a remote server? What is the frequency of server – client update?
- Is there queue maintained when one flow is triggered and another trigger for same code flow is waiting for execution? If there is an event wait queue, execution wait queue, any buffer that is used. Can this be tested?
- Do the logs collected roll over? Is roll over applicable for all event types?
- Are there UI action alerts involved? Are these actionable alerts or informational alerts? What is the queue like when multiple actionable alerts are queued?

The outcome of subjecting a software module to the above list helps us pick those basic functional tests which can target the high frequency code flows and their potential deviation paths. The goal is to pick the most basic feature and design tests to use its intended behavior under unexpected conditions.

Example:

- When a restricted action is performed on an enterprise managed machine, the event is blocked and event type is logged in a central management DB capturing system information, user details and the restricted operation performed. This information is stored in a remote DB. Can we perform restricted action on the managed machine to flood the database? Can we tweak the test, as to send more information to the server?
- Email scanners on mail exchange servers scan incoming emails for malicious file attachments. This is a basic feature. If the attachment is a compressed file, the scanner would decompress it and then scan it. This is a sub-flow. Now, if the file attached is a super compressed file, the exchange server will need to decompress it and this action can consume the entire CPU and disk space; this is a DoS attack. For the right reasons, when decompression exceeds a certain size or CPU utilization, it gets flagged as DoS. What about small, clean but nested compressed files or a Zip bomb (Peter 2004-2009)?

4.3 Introduction to Soak Testing

Once a high frequency code flow and its normal (positive) and deviation paths are identified, the next step is to subject the path for targeted soak tests. A widely accepted definition of soak testing is “soak testing involves testing a system with a typical production load, over a continuous availability period, to validate system behavior under production use.”

In simplified terms, our goal is to do high volume testing of high frequency code flow for normal and deviation events. This type of testing helps to simulate months of activity on the system in short duration of time.

Below figures explain a hypothetical case of soak testing.

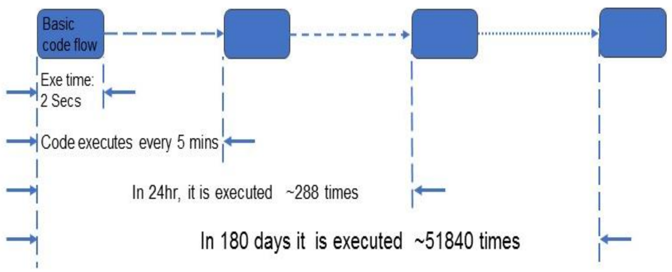

Basic code flow
Exe time: 2 Secs
Code executes every 5 mins
In 24hr, it is executed ~288 times
In 180 days it is executed ~51840 times

Figure 1 represents a basic code flow execution under a positive condition, which we would expect to be triggered approximately every 5 minutes in the production system. The code itself takes approximately 2 seconds to execute under normal operating conditions. In a 24-hour period we would expect this event to be triggered approximately 288 times in the production system.

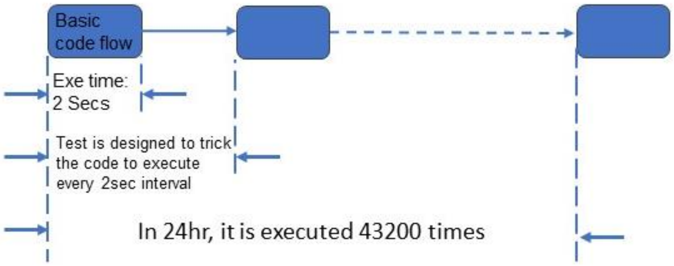

Basic code flow
Exe time: 2 Secs
Test is designed to trick the code to execute every 2sec interval
In 24hr, it is executed 43200 times

Figure 2 represents soak tests which focus on time compressing the event trigger from 5 minutes to every 2 seconds. In a 24-hour period we would expect this event to be triggered 43,200 times in our test system, simulating 150 days of production environment.

With this trend, in 36hrs we would have 64,800 executions. This could be soak test under ideal conditions.

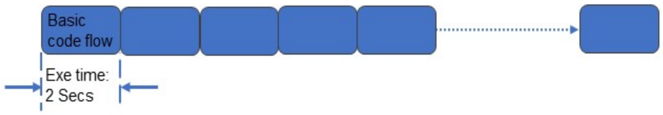


Figure 3 represents further time compression of the event by continuously triggering of the same block of code. This represents positive flow (non-deviant) soak test

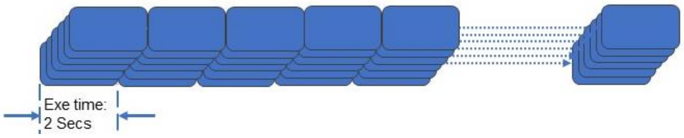


Figure 4 represents the parallelization of the continuous testing of the same positive flow of code. This test also begins to represent a positive flow, DoS simulation.

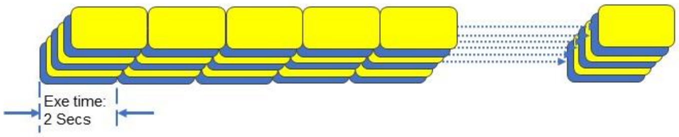


Figure 5 introduces the deviation flow test being executed in the soak test. This scenario begins the transformation from the positive flow, DoS simulation to the deviation flow, DoS simulation.

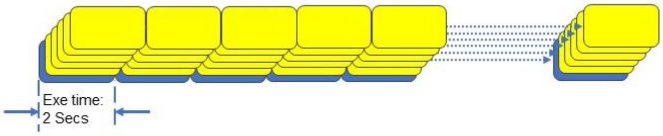


Figure 6 illustrates one positive flow, among a series of deviation flows. This is a full deviation flow, DoS simulation.

These soak variants (Figure 4, Figure 5 and Figure 6) not only help expose soak issues that Figure 2 and Figure 3 could expose but in addition also expose if there are any other hidden performance issues. This is our area of interest for this paper. In real world, code flow requests are both synchronous and asynchronous in nature. The above described approach though simplistic, still holds good for both conditions.

4.4 Guidelines for Running Soak Tests

Below are guidelines on how long to run the tests, and how to identify if this soak scenario, if extended could result in a DoS attack.

- For Windows environment, perfmon (Rusen 2016) is used to record resource utilization counters. Private Bytes and handle count is a good set to begin with. Note: Private Bytes refer to the amount of memory (RAM) that the process executable has asked for. Handle count is the number of logical associations with a shared resource like a file, window, memory location, etc.
- Duration: 25% of the time is for pre-cooling (this is a test ramp up time when SUT is subject to no load. It is expected to be in running condition and waiting for specific path trigger),100% run time, 25% of time should be post-cooling (this is the state where all the load is taken off and SUT is expected to cool down, in best case resource come to a base level as exhibited by pre-cooling time behavior). Example: 4hrs test duration, 1hr will be pre-cooling and 1hr will be post cooling. Total 6hrs.

The figures below depict hypothetical cases of both a clean soak run and a run which shows signs of resource leak.

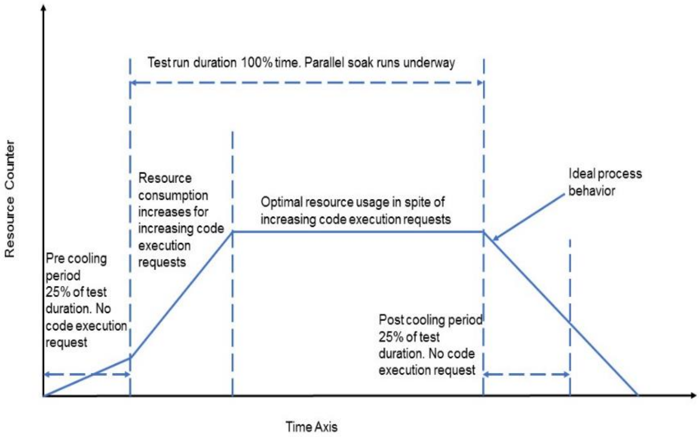

Test run duration 100% time. Parallel soak runs underway
Resource Counter
Resource consumption increases for increasing code execution requests
Optimal resource usage in spite of increasing code execution requests
Ideal process behavior
Pre cooling period 25% of test duration. No code execution request
Post cooling period 25% of test duration. No code execution request
Time Axis

Figure 7:

This illustrates an ideal process resource utilization graph.

1st spike in memory or handle consumption: When actual test starts the memory and handle counts begin to ramp until optimal resource usage is achieved. This results in a flattening of the curve. In an ideal case, the memory and handle graph for a process will flatten after increasing for about 30-40 mins of peak tests. For some processes, it may take 1-2 hrs as well.

Decreasing trend: Once the test is over, resources consumed are freed, displaying ideal behavior

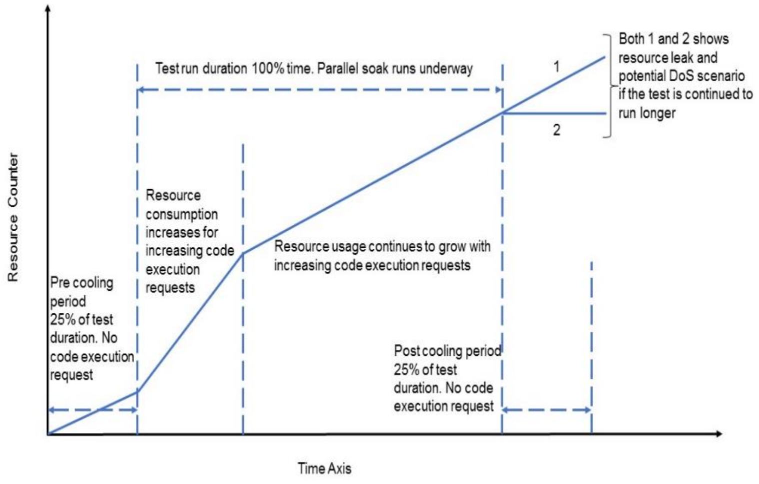


Figure 8:

This illustrates a process resource utilization graph which shows resource leaks as well as potential DoS scenario

Resource utilization keeps increasing with increasing code executions requests.

The cases mentioned (line 1 and 2) post code execution requests stop, in both cases, if the code execution requests would have continued, it would have been a potential DoS due to over utilization of resources.

Resource leak and DoS indicators:

- For Private Bytes: If we see an increasing trend in consumption of Private Byte at the rate of 500KB/hr. for more than 4hrs, it’s a concern. (This rise is monitored after initial jump in resources.)
- For handle count: 25 handles/hr. for consecutive 4hrs is a concern.
- If graph shows slow growth rate, but consistent increase, a re-run of the test with increased time duration is recommended. Probably it is a small resource leak not resulting in DoS.
- If graph shows huge jump e.g. 250 handles/hr. and keeps growing, it’s a clear concern. It is both a resource leak as well as if the tests are run long enough could result in DoS.

Reference to previous runs:

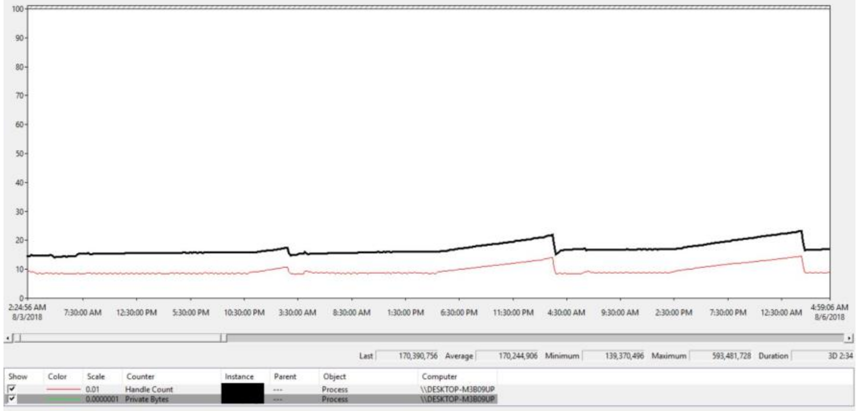
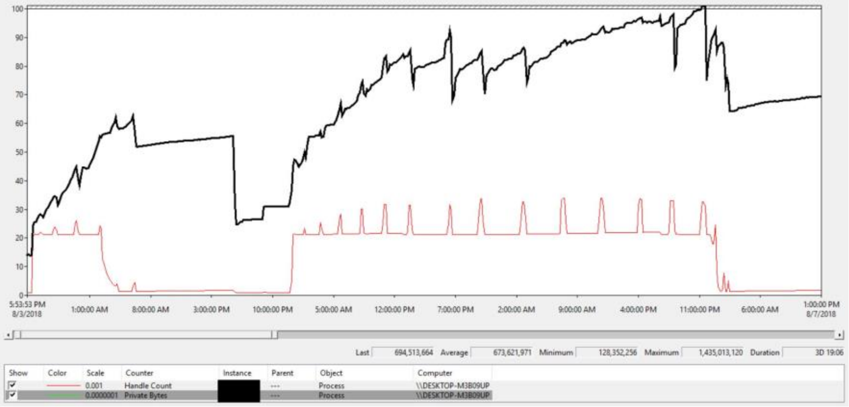
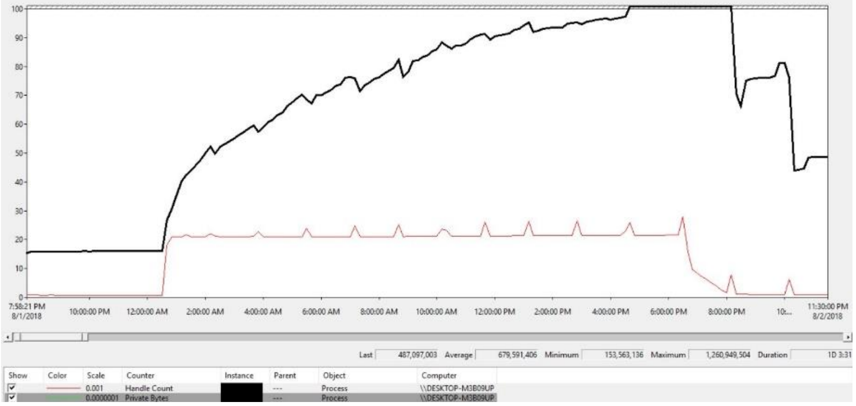
- Every test run is unique, and every operating system (OS) resource consumption is unique.
- Same test, same process on different OS’s consume resources differently. On same OS, process resource consumption comparison in different test runs is valid. e.g. OS A vs OS B memory consumption cannot be compared for same test. However, different test runs of same test on OS A vs OS A, resource consumption can be compared.

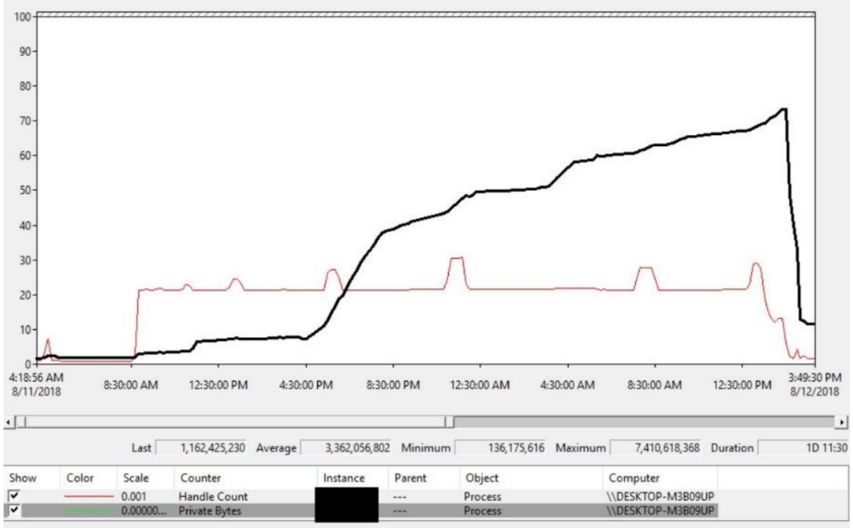
4.5 Case Study of How to Build a Soak Scenario

On a popular OS type, we created two types of files: type A and type B. Both files are text files with varying content. Each file type creation triggers specific code flow execution. Below are couple of simulation run graphs when we tweak file name for long and short names and we measure a system process private bytes.

System configuration: 8GB Ram, 8 Core, 100GB HDD

We used a sample python script to create different file types i.e type A and type B files for this simulation.

 100 90 80 70 60 50 40 30 20 10 0 2:24:56 AM 8/3/2018 7:30:00 AM 12:30:00 PM 5:30:00 PM 10:30:00 PM 3:30:00 AM 8:30:00 AM 1:30:00 PM 6:30:00 PM 11:30:00 PM 4:30:00 AM 9:30:00 AM 2:30:00 PM 7:30:00 PM 12:30:00 AM 4:59:06 AM 8/6/2018 Last 170,390,756 Average 170,244,906 Minimum 139,370,496 Maximum 593,481,728 Duration 3D 2:34 Show Color Scale Counter Instance Parent Object Computer ☑ 0.01 Handle Count [redacted] --- Process \\DESKTOP-M3B09UP ☑ 0.0000001 Private Bytes --- Process \\DESKTOP-M3B09UP Test Duration: 3Days 2hrs, Private Bytes: Max value ~565MB.	**Figure 9:** Type A files are created with maximum allowed file name character. Default maximum length of file name is 256 characters. **Analysis:** The system and application are stable with no memory leaks.
 100 90 80 70 60 50 40 30 20 10 0 5:53:53 PM 8/3/2018 1:00:00 AM 8:00:00 AM 3:00:00 PM 10:00:00 PM 5:00:00 AM 12:00:00 PM 7:00:00 PM 2:00:00 AM 9:00:00 AM 4:00:00 PM 11:00:00 PM 6:00:00 AM 1:00:00 PM 8/7/2018 Last 694,513,664 Average 673,621,971 Minimum 128,352,256 Maximum 1,435,013,120 Duration 3D 19:06 Show Color Scale Counter Instance Parent Object Computer ☑ 0.001 Handle Count [redacted] --- Process \\DESKTOP-M3B09UP ☑ 0.0000001 Private Bytes --- Process \\DESKTOP-M3B09UP Test Duration: 3Days 19hrs, Private Bytes: Max value ~1.3 GB.	**Figure 10:** Type B files are created with short file names (8-15 characters) **Analysis:** With each type B file creation, memory consumption keeps increasing and continues to grow. We do not see optimal resource utilization level reached. Private bytes show signs of memory leak.
 100 90 80 70 60 50 40 30 20 10 0 7:58:21 PM 8/1/2018 10:00:00 PM 12:00:00 AM 2:00:00 AM 4:00:00 AM 6:00:00 AM 8:00:00 AM 10:00:00 AM 12:00:00 PM 2:00:00 PM 4:00:00 PM 6:00:00 PM 8:00:00 PM 10:... 11:30:00 PM 8/2/2018 Last 487,097,003 Average 679,591,406 Minimum 153,563,136 Maximum 1,260,949,504 Duration 1D 3:31 Show Color Scale Counter Instance Parent Object Computer ☑ 0.001 Handle Count [redacted] --- Process \\DESKTOP-M3B09UP ☑ 0.0000001 Private Bytes --- Process \\DESKTOP-M3B09UP Test Duration: 1Day 3hrs, Private Bytes: Max value ~1.2GB.	**Figure 11:** Type B files are created with maximum allowed file name character **Analysis:** By stacking multiple type B file creation events along with max length of file name we have started converting Figure 10 scenario to deviation flow. Private bytes show signs of high memory leak.



Last 1,162,425,230 Average 3,362,056,802 Minimum 136,175,616 Maximum 7,410,618,368 Duration 1D 11:30

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	🟥	0.001	Handle Count	[redacted]	---	Process	\\DESKTOP-M3B09UP
☑	🟩	0.00000...	Private Bytes	[redacted]	---	Process	\\DESKTOP-M3B09UP

Test Duration: 1D 11hrs, Private Bytes: Max value ~7 GB.,

Figure 12:

Parallel soak of type B file creation with maximum allowed file name character

Analysis:

Figure 11 test is converted to a parallel soak scenario (Refer Figure 6 and Figure 8) and we see signs of high memory leak and after this point a critical service failed. DoS scenario was hit. When we restarted the system, the service fails to restart, as it is unable to load file creation log entries, SUT has broken down.

5 Parallel Soak Testing and how it is linked to Security Testing

Functional testing dominates the testing landscape with little to no scope for non-functional test areas. For Security and Performance testing, standard test templates are followed which do not cover much beyond what is traditional. Due to aggressive project time lines, security testing and performance testing has not had its fair share of attention

- **For Security Testing:**
  Standard templates or models are used, and security testing is planned. Standard tests are, threat modelling, penetration testing and to lesser percentage privilege escalation. DoS attack does not have dedicated test slot. It is, in most cases a post release test.
- **For Performance Testing:**
  Performance testing is heading towards being a check box item in overall testing life cycle. Major emphasis is given to standard test templates defined or assume positive conditions in test environment. These tests are a good start for projects doing performance testing for the first time.

The gap in performance testing gives scope, where a DoS attack could be possible by exploiting the basic built in feature of the module. When soak testing is planned, it is planned only for positive linear flow and never considers a parallel soak conditions. This switch in our approach to run parallel soak test, help us expose potential DoS scenarios and as well as soak issues. The strength of the test is to use basic functional test case, and re-wire the conditions to expose issues which otherwise are waiting to be exploited in field.

There are multiple reasons why this area of testing is overlooked or has never been exposed. Among them are:

- The level of expertise required range from advanced to expert level functional testing knowledge.
- Understanding of how to leverage functional test cases to expose alternative code path weakness.
- Having a good understanding of logging involved at each failure stage is the key to parallel soak testing. This helps to better select the most vulnerable, high frequency code flows for test.

- Time. Product release cycles are under immense pressure, but making time for this type of testing, even periodically, is the key.

6 Key Takeaway of Parallel Soak Testing Approach

Parallel soak tests exhibit high test case effectiveness. Every test case is effective until it fails, once a fix is in place, chances of this modular test case failing are greatly reduced. There have been multiple attempts to define test case effectiveness mathematically (Chernak 2001), as-well-as to define test case efficiency (Naidu, Vasudeva 2012). Both definitions can be used to justify parallel soak test effectiveness.

We would use "*Test case efficiency*" with respect to detecting defects as a closer measure to evaluate soak testing contribution to test life cycle. Our definition is "the number of defects found per test case designed".

Test case efficiency = (Total actual defects found / Number of times a test case is run) * 100

- Number of times a test case is run = Number of times a specific test case is run against the application. Minimum value being 1.
- Total actual defects found = Total Actual defects found using this test case (which is intended defects + other defects found)
- Intended defects = Intended defects to be found by each test case, which at minimum per test case, will always be 1. This value cannot be zero, simply because you write a test case to find a defect, at the very minimum 1 defect, else the test case should not exist.

Our Observations:

- A value of 0 indicates the test case never found a defect, or all defects detected by it have been fixed, hence making the code immune to this test.
- A higher number would indicate multiple defects found by a single test case. This makes the test case one of the most effective and efficient test cases to be executed on every run.
- For a well-designed soak test case, the test case efficiency will always be a greater then 1, as it not only helps to expose performance issues but also, it exposes design flaws and security issues. For a functional test case, this value ideally will always be '1'
- For every test release cycle, when we evaluate test case efficiency after at least one run, a '0' would indicate time for designing a new test case.

Given a higher "test case efficiency" factor, return of investment to uncover defects is higher in soak tests when compared to standard functional tests.

Conclusion

Targeted parallel soak tests help expose resource utilization issues and low-rate DoS vulnerabilities in applications. Traditional performance and security testing do not cover this particular application security vulnerability/performance weakness. These tests can be achieved by simple rewiring of existing functional tests and adjusting the current performance testing approach. By doing these simple adjustments to the existing testing regiments, the parallel soak testing of deviation flow has a high rate of test case efficiency and yield positive return on investment (ROI) for the organization.

7 References

Anderson, Paul. 2008. "The Use and Limitations of Static-Analysis." *CrossTalk-Journal of Defense Software Engineering* 21. https://pdfs.semanticscholar.org/e716/7b08d27f1b92c42e573d0e59bf2d2810844d.pdf.

C. Lonvick. 2001. "The BSD syslog Protocol." *tools.ietf.org.* August. Accessed July 24, 2018. https://tools.ietf.org/html/rfc3164.

Changwang Zhang, Zhiping Cai, Weifeng Chen, Xiapu Luo, Jianping Yin. 2012. "Flow level detection and filtering of low-rate DDoS." *Computer Networks, Volume 56, Issue 15* (Elsevier BV) 3417-3431. doi:https://doi.org/10.1016/j.comnet.2012.07.003.

Chernak, Y. 2001. "Validating and improving test-case effectiveness." *IEEE Software, vol. 18, no. 1* 81-86. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=903172&isnumber=19528 .

D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. 2011. "Improving software diagnosability via log enhancement." *In Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)* 3-14. http://opera.ucsd.edu/paper/asplos11-logenhancer.pdf.

Microsoft. 2018. "Get started with Windows 10." *Download and install the Windows ADK.* April 30. Accessed July 24, 2018. https://docs.microsoft.com/en-us/windows-hardware/get-started/adk-install.

Naidu, Vasudeva. 2012. *Test Case Effectiveness - What does it say?* February 14. Accessed July 20, 2018. http://www.infosysblogs.com/testing-services/2012/02/test_case_effectiveness_-_what.html.

Peter, Bieringer. 2004-2009. *Possible Denial-of-Service caused by decompression bombs.* Accessed July 15, 2018. http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html.

Rusen, Ciprian Adrian . 2016. *How to start the Performance Monitor in Windows.* December 2. Accessed July 15, 2018. https://www.digitalcitizen.life/basics-about-working-performance-monitor.

Softwaretestinghelp. 2018. *Top 40 Static Code Analysis Tools.* August 8. Accessed August 10, 2018. https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/.

Survey, Vittalkumar Mirajkar, Sneha Mirajkar, and Narayan Naik. 2018. "Study to understand Code Execution frequency." *SurveyMonkey.* July 25. Accessed August 24, 2018. https://www.surveymonkey.com/r/RKKKD52.

The Apache Software Foundation. 1999-2018. *Apache Logging Services - Log4j.* Accessed July 24, 2018. https://logging.apache.org/log4j/2.x/.

Yuan, Ding, Soyeon Park, and Yuanyuan Zhou. 2012. "Characterizing logging practices in open-source software." *ICSE 2012* 102-212. http://opera.ucsd.edu/paper/log_icse12.pdf .