

# Cyber Trust Handbook for improving security in wireless product

Business Case Secure Wireless Technology Platform in  
Work Package 3: Securing Platforms and Networks

Markku Kylänpää VTT  
Raino Lintulampi Bittium Wireless  
Pekka Pietikäinen University of Oulu  
Tero Takalo Capricode



## Table of Contents

Introduction.....	3
Integrity Protection.....	4
Utilizing Fuzz Testing in Agile Test Automation.....	15
Product security incident response team.....	26
Device Management by 2020 .....	39

## Introduction

This document is an output from the business case *Secure Wireless Technology Platform* which is part of *Work Package 3: Securing Platforms and Networks*. The objective is to give an overview of selected tasks executed in the business case during the Cyber Trust program in a more general level. Because this is the last deliverable in the business case it is also an input for the program's dissemination publications which will be finalized in the program's last phase.

The focus in this document is the improvement of the security of wireless product during the product's life cycle from development to the product's end of life. The first two articles describe issues which need to be taken into account in designing the secure product. In *Integrity protection* it is described how the software's integrity can be maintained and possible changes by malware can be detected without a delay. Integrity protection requires hardware trust anchor and mechanisms to protect both boot chain and user space components. Integrity protection ensures that the platform software's integrity is kept.

No design is perfect and less is the implementation. Therefore testing is needed and fuzzy testing has turned out to be a very effective testing method. Article *Utilizing Fuzz Testing in Agile Test Automation* introduces methods to make fuzzy testing more effective and easier to use.

Implementation is not perfect but neither is testing. This is especially true today when product software is built using hundreds of open source components. Article *Product Security Incident Response Team* describes how product software's vulnerabilities can be easily and constantly monitored until a product's end of life.

Finally the article *Device Management by 2020* addresses practical issues which should be considered in the future while there are connected devices nearly everywhere in our living environment.

## Integrity Protection

Markku Kylänpää, VTT

### *Executive summary*

Guidelines for integrity protection methods are given. Integrity protection requires hardware trust anchor and mechanisms to protect both boot chain and user space components. Linux kernel contains multiple alternatives for integrity protection. Remote attestation mechanisms can be used to provide integrity proof of a connecting client to a remote server providing service to the client.

### *Introduction*

Integrity protection mechanisms are needed to prevent unauthorized modifications to system software and applications. Sometimes this can even be a safety issue if unauthorized modification is causing danger, like radio interface exceeding permitted Specific Absorption Rate (SAR) values. Integrity protection is also a corner stone of security and it is needed to guarantee that all protection mechanisms expect to operate as designed. Whether there are complex access control or Digital Rights Management (DRM) mechanisms to protect content, those all will fail if attackers are able to manipulate components that are supposed to be trusted. Therefore, trust requires that integrity of components handling confidential information should be verified before these components are used.

Integration protection mechanisms are developed to support chained verifications where each component in boot chain is verifying the next component before passing control to it. For example, the first stage bootloader should verify the second stage bootloader and the second stage bootloader should verify kernel image. Verification is typically done by calculating cryptographic hash of the component and then verifying the result using the signature of the verified image. This verification chain requires that there is a trusted starting point. The system must have security hardware like Trusted Platform Module (TPM) or ARM TrustZone, which allows binding of device identity and the use of device specific signing keys.

System integrity is a basic building block for all trusted systems. Attackers may want to break into the system and can try to modify system software. Limiting physical and network access to the system has been a simple way to protect systems but it is not feasible approach with highly connected devices. This means that integrity protection mechanisms are needed to mitigate these threats. Integrity protection mechanisms can be split into categories:

- Root of trust
- Integrity protection of early boot and kernel

- Integrity protection of userspace
- Providing integrity proof to remote systems

These categories provide different mechanisms for integrity protection.

## ***Data integrity models***

Various formal models of data integrity can be used to formalize data integrity goals. In general, main goals are prevention of unauthorized modifications and maintenance of internal and external consistency. Biba and Clark-Wilson integrity models are two most well-known data integrity models.

Biba model assumes that users processing information are bound to certain integrity level and they can only create content at or below their own integrity level and view content at or above their own integrity level. This concept is also known as “write down, read up”. Typical clarifying example is military officers writing orders to lower officers that should follow orders of their superiors and not take orders from lower rank officers. This model does not cover confidentiality issue at all and is in fact exactly opposite to well-known Bell-LaPadula security model that is often characterized as “write up, read down”.

Another well-known integrity model Clark-Wilson maintains integrity by restricting access to integrity protected data to a small set of procedures. Clark-Wilson splits data sets to Constrained Data Items (CDIs) and Unconstrained Data Items (UDIs). CDIs can only be accessed and manipulated using Transaction Procedures (TPs). Integrity Verification Procedures (IVPs) can be used to verify that CDIs conform to the integrity constraints. Instead of classic access control matrix with subjects accessing objects, there is Clark-Wilson triple - subject/TP/object.

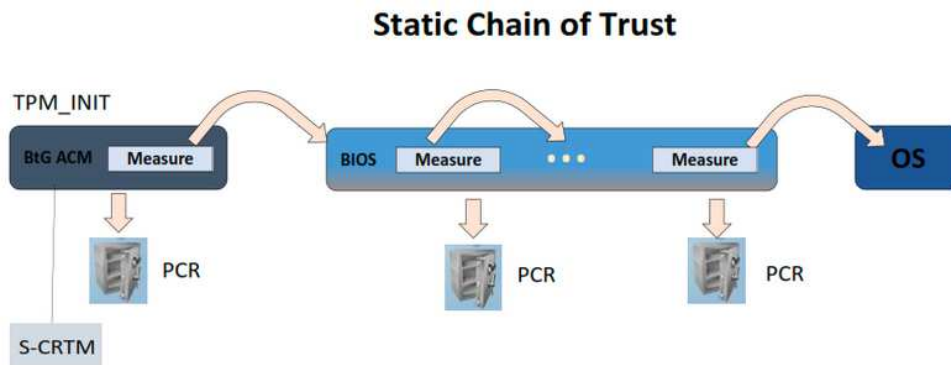
FreeBSD kernel contains implementation of the Biba integrity model as an optional component. All system subjects and objects are assigned integrity labels that are ordered. Thee implementation also assumes few special labels that can be used in situations where strict Biba model does not work.

## ***Hardware trust anchor and root of trusts***

Platform security approaches assume having a minimal Trusted Computing Base (TCB) that is stored in immutable storage. This TCB can contain boot sequence code, minimal crypto library. Also small set of data like trust root (e.g. hash of the public key) and device identity (e.g. serial number).

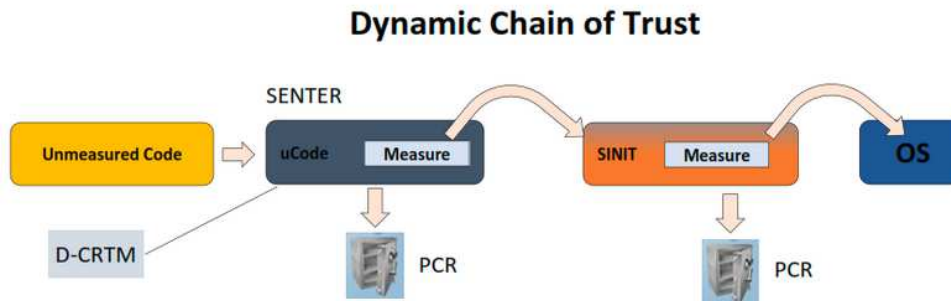
Integrity protection mechanisms typically assume that there is a minimal immutable initialization code often called the “Core Root of Trust for Measurement” (CRTM). The code is executed after system reset from immutable storage e.g. boot ROM and should always behave in the expected manner.

“Static Root of Trust for Measurement” (SRTM) is a mechanism defined by Trusted Computing Group (TCG). After system reset each component in a boot chain is measured by the predecessor code before control is passed to the component. Measurement value is then extended to one TPM Platform Configuration Register (PCR).



**Figure 1 Static Root of Trust for Measurement (Source: [blog.fpmurphy.com](http://blog.fpmurphy.com))**

“Dynamic Root of Trust for Measurement” (DRTM) is a mechanism supported by Intel and AMD. Intel Trusted Execution Technology (TXT) and AMD Secure Virtual Machine (SVM) technologies contain platform enhancements and extra processor instructions to allow the launch of measured environment without resetting the system.



**Figure 2 Dynamic Root of Trust for Measurement (Source: [blog.fpmurphy.com](http://blog.fpmurphy.com))**

### ***Boot integrity protection***

Integrity protection should start at boot time. Boot process can be constructed so that each component verifies the next component before passing control to the next component. This chained verification mechanism also requires reference values to verify integrity. Reference value is typically a signature that is signed using a certified key. If verification fails, there are two options. Boot process can be terminated. Alternatively, boot process can continue but user is notified that verification has failed. Some system allow measurement of components using security hardware e.g. TPM.

Booted kernel should also include protection against code loading. Either the kernel should be static, without support for loadable kernel modules, or all loadable kernel modules should be signed. The kernel modules must be verified before loading.

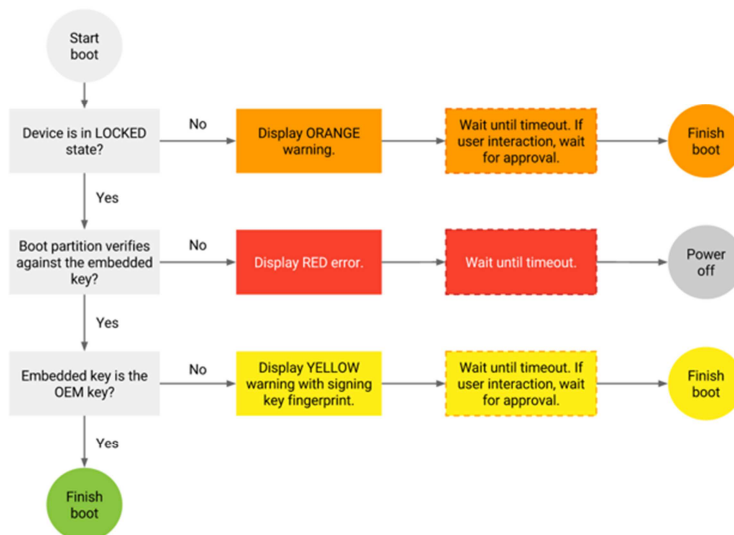
## Secure boot

A boot sequence starts execution from a fixed code that is typically called a boot ROM. The boot ROM then loads a boot loader code that starts kernel. There can also be a chain of boot loaders for different boot stages. Open systems have no restrictions for code loading. Closed systems have closed boot loaders that only load authorized software.

Secure boot requires that the loaded code should be authorized to execute. This can be achieved by requiring that there should be a matching signature of the loaded code that can be verified. The public key to be used should be verified already in boot ROM. For example, the boot ROM could calculate a cryptographic hash of the public key and compare it to an immutable reference value. Alternatively, the key itself can be a part of the boot ROM. Secure boot systems have closed boot loaders that only boot authorized software. If verification fails, the system will not boot.

## Verified boot

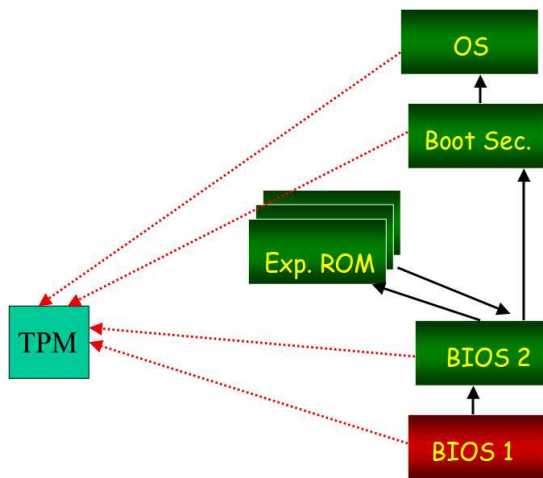
Verified boot is a variation of secure boot that can be less strict and can allow end user to make a final boot decision in cases where non-standard system state or software is detected. For example, Android has a boot colour concept (see Figure 3) that is used to notify users about non-standard system configuration. However, even in this case the device manufacturer has many control points. If the device manufacturer does not provide mechanism to unlock the device and embed extra OEM keys then verified boot mechanism can be very similar to secure boot.



**Figure 3 Android Verified Boot (Source: Google)**

## Authenticated boot

Authenticated boot (sometimes also known as measured boot) is a mechanism where boot chain stores measurements of the launched components to integrity-protected storage. Typically, Trusted Platform Module (TPM) Platform Configuration Registers (PCRs) are used. There are TCG specifications that specify what components are measured and what TPM PCRs are extended to store these measurements. Authenticated boot is passive boot method. Neither user nor components in boot chain make any boot continuation decisions. Boot chain components are just measured and measurements can later be used to prove system state.



**Figure 4 Authenticated boot (Source: William A. Arbaugh)**

## Trusted boot

Trusted boot also measures all boot stages but also implements secure boot functionality allowing only authorized components to be loaded. Trusted boot is a combination of authenticated boot and secure boot. It should always boot into trusted state but it can also provide measurements that can be used to attest boot state.

## *Userspace integrity protection*

### Introduction

Boot integrity is crucial as modification of boot components allows attackers to control kernel and kernel has access to all data in systems. However, also attacks against user space components can be used to gain control of systems. Attackers that are able to replace system services can also gain access to confidential information.

Secure boot concept often verifies only various boot loaders and booted kernel. Small systems can afford verification of the whole system image as large binary blob before executing the first binary executable (*init* program in Linux). However, as the size of the system image increases this approach becomes non-feasible, as verification of large image



will slowdown boot. Other methods that interleave verification with execution have been developed. Main purpose of the integrity verification is to protect against offline attacks.

## Linux Integrity Measurement Architecture (IMA)

### Integrity measurement

Instead of verifying everything at once as one blob, it is possible to verify files when files are first time used. Integrity Measurement Architecture (IMA) subsystem in Linux kernel can be used to take measurements of executable files and by default also all files read by the root user. If the device has TPM, the measurement is also extended to one of TPM PCRs. Remote verifier can then request attestation and check status of the system.

IMA provides a measurement list of all measurements taken. All components are measured only for the first time and will not be measured again unless the file is modified. The list is also available to userspace as Linux kernel securityfs entry.

PCR	SHA1 template hash	Template	SHA1 file hash	Filename
10	9f4dee7500339f81373ffc7ef1588ee04ff348cc	ima-ng	sha1:9797edf8d0eed36b1cf92547816051c8af4e45ee	boot_aggregate
10	0beladed7ca430f3a823ffa0c983bd2f3f91ecde	ima-ng	sha1:b51a9c6c6955ec818dae60a15554c80c891c99227	/etc/preinit
10	c51dc35733735dd17d2d7459cd28fc08d66cfcdd8	ima-ng	sha1:c3c199e5ba3706b0ee2937266f6da71c2e13a63f	/bin/busybox
10	d3a4e796559b5d53e240f01cabd4534a6e49ad18	ima-ng	sha1:ed9c9db11fc54ac66e0a8c0ee344bc67b4ac9774	/lib/libc.so
10	a96af84d01931b3d956251792988fd87260cac61	ima-ng	sha1:19bacb1637249596f4d11fdb0d4a94d1d27c498c	/lib/libgcc_s.so.1
10	a372aa9516827d8c03cce913dc6de412739401a0	ima-ng	sha1:c6ab0fc6f6347bf7808ab48b4e951958f925274f	/sbin/init
10	6abel18d7f83c801ec8f2ba1e15826bbd785b79d0	ima-ng	sha1:41404f2c907da4807770b055484dfd544290b566	/lib/libubox.so
10	49eaaeb81f0d2a36456bd15f3287edca171a62e1	ima-ng	sha1:7142db81e3bf7ea0f502e28266b04464c4bc61fa	/lib/libubus.so

Figure 5 IMA measurement list example

IMA measurement mechanism is policy driven. There is a built-in policy that can be enabled but also custom policy can be later loaded using securityfs file interface. Policy can be used to specify measurements when IMA specific hooks are triggered in kernel. Either “measure” or “dont\_measure” rules can be used. BPRM\_CHECK rule enable measurement of applications and scripts that are started as commands. FILE\_MMAP rule can be used to trigger measurement of shared libraries. The third rule FILE\_CHECK triggers measurement when a file is opened.

The rules can have additional specifiers that could limit the rule for certain operation e.g. MAY\_READ for reading operation. There are also file system type specifier (fsmagic) that can be used disable measurements from pseudo file systems like procfs, sysfs, tmpfs, and securityfs. In addition, security module labels (e.g. SELinux) can be used in rules.

```

action: measure | dont_measure
condition:= base | lsm
    base: [[func=] [mask=] [fsmagic=] [uid=]]
    lsm: [[subj_user=] [subj_role=] [subj_type=]
        [obj_user=] [obj_role=] [obj_type=]]

base: func:= [BPRM_CHECK] [FILE_MMAP] [FILE_CHECK]
mask:= [MAY_READ] [MAY_WRITE] [MAY_APPEND] [MAY_EXEC]
fsmagic:= hex value
uid:= decimal value
lsm: are LSM specific

```

Figure 6 IMA policy language

### Integrity appraisal

IMA measurement mechanism itself does not provide any protection but unauthorized modifications can be detected if files are measured and measurements are compared to

known good whitelist values. There is also local verification mechanism that is utilizing IMA measurements. IMA appraisal concept compares stored measurements into reference values that are stored as extended attributes of the file. Extended attribute *security.ima* is used. The value should be integrity protected by either using HMAC or digital signature. IMA appraisal mechanism is policy driven. IMA policy can contain “appraise” and “dont\_appraise” rules using the same policy language as IMA measurement mechanism. IMA and EVM policies are loaded together.

## *Extended Verification Module (EVM)*

Linux security frameworks SELinux or Smack use their own extended attributes that should be integrity protected. Extended Verification Module (EVM) can provide protection to these attributes by adding a new extended attribute called *security.evm* that contains signed hash over these attributes. Extended Verification Module (EVM) can be configured to protect the following extended attributes if they exist:

- ***security.ima*** – IMA reference hash values.
- ***security.selinux*** – SELinux file object labels
- ***security.SMACK64*** – Smack security module file labels
- ***security.SMACK64EXEC*** - Smack security module file labels
- ***security.SMACK64MMAP*** - Smack security module file labels
- ***security.capability*** – File system capabilities

EVM is using either HMAC or digital signature. Optionally also filesystem identifier can be included so that extended attribute cannot be copied from other filesystem.

## *Strengths and weaknesses of the IMA approach*

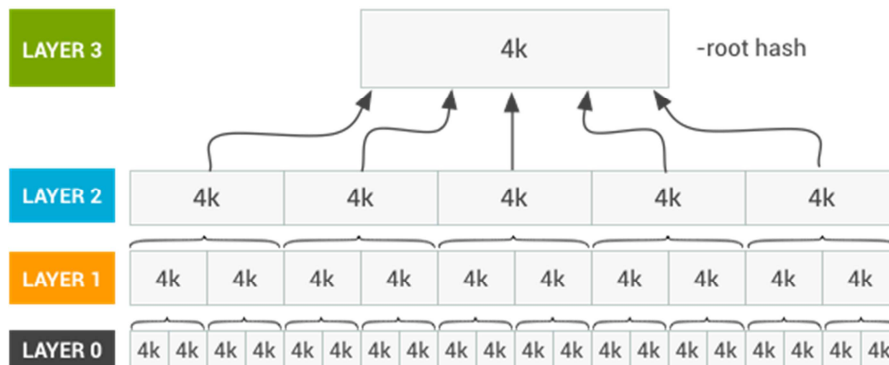
File-based verification has many problems. File-hierarchy pathnames can be manipulated by using hard links and mounts. In addition, directory content should be verified as sometimes also removing files can be used to manipulate the system. For example, if configuration files are read using search path, and if the first file is removed, configuration settings are then read from the next file found from the search path. The configuration may be different allowing attacker to gain access to the system.

Another problem is that large files are verified although only part of the file is mapped and used. This could be a problem if large media file or very large shared library is measured. However, as the measurement is only done once this is a problem only during boot time.

Main strength of IMA is that it provides a measurement list that can be used in remote attestation.

## *Dm-verity*

As IMA is measuring files the whole file must be read into memory before it is verified. Sometimes this can be slow and unnecessary. Sometimes only small part of large libraries or verified multimedia files should actually be executed or read. The IMA approach forces the whole file to be read but fortunately only during the first execution/reading. Block based verification alternative does not suffer from this limitation. Linux kernel has a block based verification mechanism called dm-verity. Dm-verity is limited to read-only volumes. There is also unofficial patch set called dm-integrity that is supposed to work also with read-write filesystems. Dm-verity requires separate storage location for integrity reference data.



**Figure 7 dm-verity hash tree (Source: Google)**

Dm-verity hash tree is created by first calculating SHA256 hash of each 4kB block. These hashes are then concatenated and split into new 4kB blocks. SHA256 hashes of these new blocks is calculated and new next layer blocks are created. The process is repeated until there is just one SHA256 hash called root hash. This hash value is then signed.

### *Encrypted filesystem*

Conventional encryption modes used with full-disk encryption (e.g. AES-XTS) can only provide confidentiality but no support for integrity verification. Attackers may try to modify encrypted content and be able to affect to system behaviour as modifications are not detected. Although embedding own code would be difficult, the attacker could still find ways to modify the system so that e.g. some server is crashing.

The use of authenticated encryption could be used to detect offline modifications. Authenticated encryption modes (e.g. AES-CCM, AES-GCM) include also MAC calculation to encryption process and MAC verification during decryption process. Decryption key must come from secure storage or it should be derived from the passphrase that is requested from the user. Drawback of this approach are performance penalty and need to store additional integrity data. Many algorithms also require non-predictable Initialization Vectors (IVs).

### *Signed binaries*

As many systems do not utilize extended attributes all protection mechanisms that rely on extended attributes require many modifications to the system. In addition, block-based integrity is only feasible for read-only module and requires also additional raw partition to store reference values. If the target is to protect only native executables then one option is to add signature to executables. The signature can be added as new section to ELF file header and the kernel can contain verification code. Signed files must be verified so that the signature part is either skipped or treated to contain constant value (e.g. zeros) during hash calculation.

### *Local verification against stored white list*

MeeGo security framework called Mobile simplified security framework (MSSF) also included integrity protection subsystem called Validator. Instead of binding signatures to a separate ELF header section all cryptographic hash values can be added to a list with corresponding pathname. The list should be signed and software installer could update the list. The list will be loaded into kernel during boot and pathnames are replaced by inode numbers so that corresponding hash value can be found in kernel when the file is executed.

## Software updates

### Introduction

Software update mechanism is needed, as there is a need to add new functionality or fix detected problems. The updates either fix functionality problems or detected security vulnerabilities. As system software often is a combination of open source software and third party Commercial off-the-shelf (COTS) software, used in multiple systems, there may be many known vulnerabilities and also many fixes waiting to be updated to the devices. Smooth, fast and efficient software update mechanism is a crucial protection against new threats as vulnerable software versions can be utilized by attackers. However, update mechanism can also be a source for many problems as attackers may be able to misuse it to import malware to system. All updates must be verified before installation Both code integrity and source authenticity must be verified.

### Firmware blob updates

The simplest update mechanism is to update the whole system image by flashing new firmware image that contains updates to fix vulnerabilities and functionality problems. When the size of the system grows also the update package is larger. The firmware blob should be signed image and system should verify the image before flashing it to persistent storage. Storage must be separated to immutable read-only area containing system software and writable area containing data. This update mechanism is suitable for small systems but can be also used for larger systems as reinstall.

### Software package updates

Another common approach is to group software components to software packages that can be installed, removed, and updated. This will reduce the size of the update as only those packages that are changed must be updated. Package manager should have a list of software package versions and also package dependencies and potential conflicts. Updating read-only volume requires that the volume should first be remounted as read-write volume for the update. Also all software packages must be verified before installation.

Most packaging formats and systems support signed packages and software source can be restricted to well-known official repository (e.g. Google Play, Ubuntu official repositories). Open systems also support side loading so that also non-repository packages can be installed.

### Binary diff updates

If system software image is used as a read-only blob then one alternative for update is to send a binary diff file that can be used to patch the blob so that it contains updates. The use of binary diff will reduce the size of updated data so that it is feasible to download updates using also slower channel. These updates are often called Over The Air (OTA) updates. As in other cases, also in this case the origin and integrity of the update must be verified before installation. The update should also identify the software version that is required and installation to other versions should be prevented.

### Live kernel updates

System updates typically require at least stopping and restarting of those services that are updated and reboot is often needed. System providers try to minimize reboot requirements but so far update of system core components has required reboot. Reboot requirement may cause users to delay software update to more appropriate time leaving them vulnerable to

attacks. So far, at least kernel updates have required rebooting of the system. However, newest Linux kernels can be updated without rebooting the system.

## Downgrade prevention

Many systems prevent reinstallation of old software versions if user has already installed newer software version. The old version may contain vulnerabilities that are already fixed in the new version. Sometimes users want to misuse these vulnerabilities e.g. if those vulnerabilities allow unauthorized access to DRM protected content. The system could store the version number of the last installed software version and prevent to rollback older software version.

## Software updates

### Introduction

When accessing remote services it is useful to be able to provide a proof of integrity to a remote service. Service provider might have requirements, policies, and different service classes for connecting clients.

Attestation is a process of validating integrity of a computing device. One way to do this is to record measurements when program code is executed. These measurement records should then be stored in integrity protected storage. Attestation may be done locally so that the measurement is compared to a stored reference integrity value and typically code execution is prevented if the measurement does not match to the stored reference value. Another alternative is a remote attestation where a remote verifier wants to have integrity guarantee of the connected device. The process includes transferring integrity measurements from the connected device to the remote verifier with proof of origin and freshness.

### Security module

Attestation systems must have some kind of security module that can store keys, perform signing operations, and integrity protect measurements. Remote attestation requires that a reply message to an attestation request should contain a proof of origin. This can be achieved by signing the reply message with an asymmetric key that is stored in security module and whose public part is certified. Also integrity of the measurements is best achieved by using security module to protect integrity and of measurements. TPM is typical security module in PC devices. Mobile devices could use e.g. ARM TrustZone-based approaches.

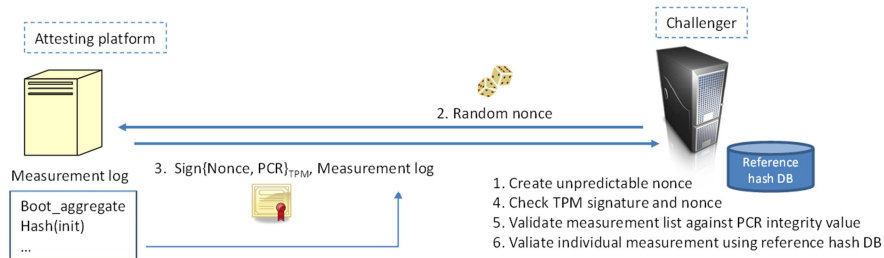
### Measurements

Attestation measurements are typically taken by calculating cryptographic hash of the content. SHA1 algorithm is still typically used but newer specifications allow the use of other cryptographic hash functions as well. Measurements are stored in the measurement log. TPM-based systems also use the so called TPM\_extend operation to store a hash chain to a register of the security module called PCR.

### Attestation protocol

Attestation request message should contain a nonce value that is used to protect freshness. The reply message should contain a signed blob with the nonce value and integrity proof of the measurement log which is the value of the PCR. The reply message also contains the

measurement log that does not have to be signed. Integrity of the measurement log can be verified using the PCR value.



**Figure 8 Remote attestation**

## Verification

The first part of the verification is to verify the signature of the signed blob and to check that the message has been signed with the certified key that has not been revoked or expired. The verification phase also requires that the verifier recalculates the PCR value that is received as part of the reply message. After these operations the individual measurements should be compared to known reference values.

## Conclusions

Integrity protection requires mechanism to protect both components of the boot chain and userspace components. Hardware-based trust anchor is needed to establish root of trusts for measurement. Linux kernel provides many alternatives to integrity protection. Either file-based or block-based approach can be used. Software update mechanism should also be protected as it can be misused to install malware to the system. Remote attestation can be used to provide integrity proof of the system to remote verifier.

## Further reading

N.Asokan & al., *Mobile Trusted Computing*, Proceedings of the IEEE, nro 8, August, pp. 1189-1206, 2014.

Will Arthur, David Challener: *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*, ISBN-13: 978-1430265832, Apress, 2015

Global Platform, *GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide*, [Online]. Available: <http://www.globalplatform.org/mediaguidetee.asp>.

David Safford, Serge Hallyn, Dmitry Kasatkin, Reiner Sailer, Mimi Zohar, Integrity Measurement Architecture (IMA), IMA wiki site, URL: <http://sourceforge.net/p/linux-ima/wiki/Home/>

J. Corbet, *dm-verity*, Linux Weekly News, September 19, 2011, [Online]. Available: <https://lwn.net/Articles/459420/>

## Utilizing Fuzz Testing in Agile Test Automation

Pekka Pietikäinen, University of Oulu

This chapter is based on "Steps Toward Fuzz Testing in Agile Test Automation" authored by Pekka Pietikäinen et al. in International Journal of Secure Software Engineering, vol 7(1). Copyright 2016, IGI Global, www.igi-global.com. Posted by permission of the publisher.

### *Executive Summary*

Including and automating secure software development activities into agile development processes is challenging. Fuzz testing is a practical method for finding vulnerabilities in software, but has some characteristics that do not directly map to existing processes. The main challenge is that fuzzing needs to continue to show value while requiring minimal effort. A key question is the availability of the skills required to setup an effective fuzz test campaign. The Software Security Group is pivotal, but eventually testing will have to be done by developers with the proper tools and skills. Setting up ways of working to make fuzzing more effective is one way of doing this. We present experiences and practical ways to utilize fuzzing in software development, and generic ways for developers to keep security in mind.

### *Introduction*

Software security vulnerabilities are not a new phenomenon. With the increasing amount of digitalization in society, the impact of vulnerabilities becomes more threatening. For example, web browser vulnerabilities have resulted in widespread intrusions and the existing exploit kits continue to be a cost-effective means of installing malware. Techniques such as data execution prevention (DEP) and address space layout randomization (ASLR) make exploitation of bugs more difficult than before, and sandboxing techniques are becoming widely used means of isolating code that processes potentially malicious data. While these advances help in making exploitation significantly more difficult than it was a decade ago, they do not make it impossible. The problem of finding and fixing implementation level security issues has remained largely unchanged from what it was a decade or even two ago, mainly due to the same programming languages being used for writing software.

Fuzz testing (fuzzing) is a practical way of testing software for security vulnerabilities arising from processing external input and is usually included in secure development lifecycle models. In 2015, nearly all high-impact vulnerabilities with a Common Vulnerabilities and Exposures (CVE) entry were found with fuzzing. Fuzzing is clearly something that should be done when developing secure software, but applying it efficiently as a part of an agile process can be challenging.

A major challenge comes from test automation. Continuous Integration and Delivery (CI/CD), Test Driven Development (TDD) and Behavior Driven Development (BDD) all aim to provide mechanisms for delivering working software quickly, in small increments. Fuzz

testing campaigns do not directly map to the used workflows, and thus can easily be seen as a complicating add-on by developers.

Another difficulty comes from lack of suitable personnel. Teams need to test, instrument and correctly interpret the results of fuzzing in a multitude of environments, test legacy code, previous features as well as new functionality. Including a security expert in each development team is infeasible. To overcome this, organizations often have a central “Software Security Group”, which can be overburdened and difficult to scale to the needs of different project teams. In this paper, we describe experiences in utilizing fuzzing as a part of a software development process, and present practical ways to utilize fuzzing in software development, and present generic ways for developers to keep security in mind.

## ***Fuzzing***

Fuzzing generates invalid or random inputs and injects them into a program. This technique can be used as such with very simple instrumentation to perform black box testing, usually combined with relatively simple heuristics for detecting obviously erroneous program states, such as fatal signals, or in a more white-box fashion, such as runtime program tracing to gain insight, which can be used to control the data generation.

One of the important properties of fuzzing is that one can start testing a program with very little knowledge about the target, and gradually refine the testing by improving the test cases and instrumentation. As an additional benefit, each found issue comes paired with a proof of concept input that can trigger the error, which proves that the bug can be triggered externally. This is unlike static analysis tools, which locate bugs in software, but do not easily demonstrate how the affected code can be reached in practice. This is especially important for programs like web browsers, where nearly all input is from an untrusted, potentially malicious, source.

Fuzzing consists of the following phases:

1. Identify target
2. Identify inputs
3. Generate fuzzed data
4. Execute fuzzed data
5. Monitor for exceptions
6. Determine exploitability

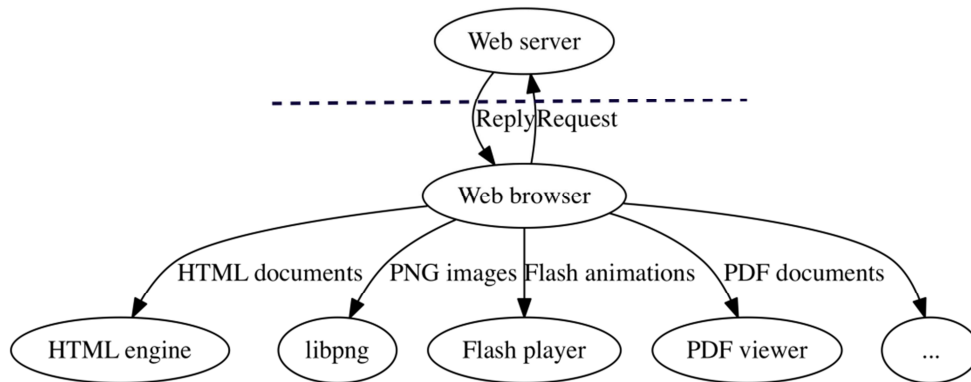
These phases can roughly be mapped to agile software development practices, which we will now describe.

## ***Identifying attack surface through threat modelling***

The target, obviously, is the software under development or an external dependency of it. The targeted component can typically be reached through a number of interfaces, such as “HTTP request over network” or “Read PNG file from local file system”, each using some more or less defined protocol. These are identified during threat modelling. Threat modelling can be done using a variety of methods, e.g., the Microsoft STRIDE model.



An example of a DFD that could be used as a basis for fuzzing is shown in Figure 1, which shows how a web browser interacts with other components. It sends requests to web servers. The server responds with content the browser processes internally or using external libraries or plugins. The content crosses a trust boundary, from the untrusted Internet to the browser process, which has access to, e.g., operating system facilities and data owned by the user who runs the browser. A more complete threat model would show these, as well as sandboxes used to isolate components that are known to have risks, such as external plugins.



**Figure 9 Simplified Threat Model of Web Browser**

Threat modelling produces a list of interfaces to fuzz, which can be prioritized and placed on the product backlog.

### **Test case generation and execution**

There are several approaches to test case generation in fuzzing. It is usually divided into two categories: generation-based fuzzing and mutation-based fuzzing, both of which have benefits and drawbacks.

Generation-based fuzzing is based on a fixed model of the input space, which is then used as a basis for building test cases. In some cases, this can be done directly from source code of the software to be tested, but often must be done separately. The method works very well with protocols and file formats that have a formal specification, from which the model can be inferred. While the model is often tree-structured, as are the protocol definitions, the model can also include fields, such as checksums, which are used by implementations to determine whether the input is valid. One of the important factors of exact models is that essential fields such as these can be automatically filled correctly making it much more likely that the target program does indeed process the contents of the test case instead of just dropping it as invalid. The main downside of generation-based fuzzing is that developing a comprehensive test suite including all the models is a major manual effort, as the test suite is essentially a minimal implementation of the software to be tested.

If the specification changes during the project, as it is often the case in agile projects, so must the model. Therefore, they are mainly of use for parts of the project where the model

remains constant, or which are considered critical enough to justify building or purchasing a test suite.

Mutation-based (or template-based) fuzzing operates based on samples of data produced by valid implementations. Most mutation-based fuzzers operate by making simple changes to the data without any knowledge about the semantics. Typical mutations include flipping bits at random, omitting data, repeating data and writing random data somewhere. Even though they lack the finesse of generation-based fuzzers, these black-box tools are important due to their ease of use and the fact that they can be made general purpose. Even the drawback of not understanding the exact protocol semantics can be worked around, e.g., testing against a system under test (SUT) that has checksum verification disabled makes it possible to improve test coverage without adding support for calculating the checksum into the fuzzer.

Radamsa<sup>1</sup> is a collection of mutation-based fuzzers, which has the primary objective of being as simple to use as possible. Unlike most similar tools, it is intended to work for just about any kind of data without any extra configuration. It does some simple heuristics on the input data (binary or ASCII), and the operation of which ranges from trivial common mutations to more novel ones, which often approximate the operation of generation-based ones.

When the best fuzzing strategy for a given interface is identified, a task to automate it can be placed in the backlog. Depending on the interface and chosen tools, this task may take anything from minutes to months. The simplest programs to fuzz are command-line based tools taking an input filename as a command line argument. These only require a fuzzer and some sample input files to get started. On the other end of the spectrum, the tested interface may be an encrypted network connection where a stateful cryptographic handshake is needed to reach a state where the test case can be injected. In this kind of scenario, sample collection, injection and test automation (or building a model) become much more involved.

### *Instrumentation and analysis*

The aim of fuzzing is to find undesired behaviour in software. Crashing is obviously such, but the effects of fuzzing can also be very subtle. An example of this is the Heartbleed vulnerability, where the response to a fuzzed test case was an abnormally large reply, which contained sensitive information, such as private keys.

For the common case of detecting crashes, building a test oracle is relatively simple, as current operating systems and compilers provide good facilities for it. The ideal output is an uncorrupted crash dump, which pinpoints the exact location in the code where the problem occurred. The fuzzed test case could cause severe memory corruption, so the program should notify the user as soon as anything abnormal occurs. Memory debugging tools, such as AddressSanitizer, MemorySanitizer and Valgrind help to ensure that the error is detected once it occurs, before corruption occurs. These tools are designed to reveal different types of errors, and have different impact for the performance of target program, so the possibility of using different instrumentation tools during testing should be considered.

---

<sup>1</sup> <https://github.com/aoh/radamsa>

For detecting other types of failures, a model of SUT behaviour is needed. For fuzzing purposes, the most trivial approach is valid case instrumentation, where a valid operation is performed on the interface. If the interface does not reply with a known valid reply, the test case is assumed to have caused a failure. Unfortunately, this approach only finds the most catastrophic failures.

When a suitable test oracle is developed, the question of integrating it into the software development process remains. Undesired behaviour needs to be transformed into an issue, which is fixed by a code commit. Furthermore, a regression test is added as a unit test, and the fix has to be released and communicated to customers.

For a static model-based test suite this is not an issue, i.e., “Test #6423 fails” maps to commonly used workflows. If the input has a random element, as in mutation-based fuzzing, the same issue can most likely be found with several similar inputs. The root cause is the same, so only one issue should be filed. The issue should contain a fully reproducible, minimized, test case. The impact of the defect can be automatically analysed (NULL pointer dereference, stack overflow), but the results are not completely reliable.

### ***Fuzzing as a part of Agile Secure Development Lifecycles***

In response to the growing impact of software vulnerabilities, a number of efforts to improve software development processes to mitigate the effect of vulnerabilities have emerged. Each company has its unique set of business, technology and cultural constraints, which affect the set of activities that are the most beneficial.

Building Security In Maturity Model (BSIMM) is a framework for measuring software security initiatives. Instead of being a guide for secure development, it is based on real-world data from sixty-seven real software security initiatives from a variety of companies. BSIMM can be used to determine the maturity of a given initiative. In BSIMM, fuzz testing is considered as an intermediate (Level 2) or advanced (Level 3) topic in security testing. (McGraw, Migués, & West, 2013)

OpenSAMM (Chandra, 2009) and SAFECode Fundamental Practices (SAFECode, 2011) are prescriptive frameworks for measuring the maturity of security initiatives. OpenSAMM does not explicitly cover fuzzing, but includes automated testing tools as a Security Testing activity. SAFECode provides specific guidance on fuzzing, including suggestions for tools.

The drawback of the previously presented approaches is that they give very little detail on how to introduce fuzzing, and how the introduction of fuzzing is seen inside the software development organization. A common minimum requirement is a data flow based threat model, which serves as a risk-based list of interfaces for fuzzing. Several hundred thousand cases are needed to have basic assurance on the robustness of each interface. If test case injection is slow, even this may be difficult to justify. Finally, evidence that testing has been done for each software increment is required.

In the next section, we describe practical experiences related to the introduction of fuzzing, which answer some of these drawbacks.

## *Practical experiences*

In this section, we describe practical experiences associated with fuzzing. We do this from the viewpoint of a research group, whose purpose is to study, evaluate and develop methods of implementing and testing application and system software in order to prevent, discover and eliminate implementation level security vulnerabilities in a proactive fashion. To do this, we have collaborated with industry by providing them with information on vulnerabilities their products have, as well as tools, whereby they could find the vulnerabilities themselves. Some collaboration efforts have lasted for several years, and the companies have shared their internal experiences with us, which we summarize (anonymously) here.

### Initial response to fuzzing

By far, the biggest hurdle has been getting started. The first experience of fuzzing for many companies is receiving an external bug report that includes a test case (possibly a stack trace) that claims to be a security bug. There is no mechanism for easily verifying the bug or finding the relevant developer based on the test case alone. Often, the severity of the bug is downplayed – a working exploit would be required to treat the bug as critical. Eventually, the bug is fixed and internal interest for fuzzing arises. We share experiences in using our tools and demonstrate how we found the bug using them.

Tools, such as Radamsa, have been built to be as easy to use as possible. If the target does not utilize files as input, a test harness and sample collection need to be implemented, which can be a non-trivial task. This may sometimes prevent fuzzing from being used.

Running a fuzzer against code that has not been fuzzed before typically finds a large number of bugs. With legacy code bases, even a quick run may find too many issues to realistically fix in a reasonable amount of time. Going through the large number of bugs requires significant effort from skilled personnel. Eventually, a business decision of fixing the most obvious ones is made, and less critical ones are simply accepted.

Initial fuzzing runs may also uncover only a few bugs. This could be due to high code quality, but also bad test coverage (e.g., due to test cases being dropped because of an incorrect checksum, poor quality samples being used with a mutational fuzzer, the SUT masking all exceptions or lack of instrumentation). Again, skilled personnel are needed to ensure that testing was useful.

### Automating fuzzing

After the initial results of fuzzing have been incorporated, the next hurdle is automating it and making it a regular part of the software development process. An individual “fuzzing expert” from the team or the SSG often builds the infrastructure. The automation is used for a while, and eventually stops finding new bugs. The expert already has new tasks and the infrastructure is eventually abandoned. In our experience, improving the fuzzer or sample set, or simply recompiling with a memory-debugging tool, would continue to find bugs.

After the initial results of fuzzing have been incorporated, the next hurdle is automating it and making it a regular part of the software development process. An individual “fuzzing

expert” from the team or the SSG often builds the infrastructure. The automation is used for a while, and eventually stops finding new bugs. The expert already has new tasks and the infrastructure is eventually abandoned. In our experience, improving the fuzzer or sample set, or simply recompiling with a memory-debugging tool, would continue to find bugs.

## Best practices

What has worked is placing fuzzing activities (automating the testing of some interface) on the backlog. Fuzzing also can leverage infrastructure from other activities, e.g.,

- UI tests done with Selenium webdriver are used with a fuzzing proxy
- The team documents all new JSON API's in a standardized way. API documentation is automatically generated and fuzzed test cases are automatically generated from the model

Agile methods require a “Definition of done”, a list of criteria that must be met before a task is considered complete. Defining a number of test cases or the amount of time used for fuzzing is a mandatory minimum, but does not ensure that testing is effective. Coverage-based techniques can be used to find areas that are not tested, but do not ensure effective testing. However, in our experience, coverage-based sample set optimization greatly increased the efficacy of mutation-based fuzzing.

Finally, the question of the economics of fuzzing and test automation is always relevant. How much resources should be spent on fuzzing and when are we doing enough? What is the opportunity cost of fuzzing, i.e., are would the effort spent on fuzzing be more productive elsewhere? Is it worthwhile to do it in-house, or can it be externalized to a bug bounty program or an external consultant?

We now present an example on how a very mature fuzzing process can be used to find defects.

## Example: Web browsers

Web browsers are the primary means of accessing Internet-based services. Their install base is in the range of hundreds of millions and the market is shared by only a handful of vendors. In the beginning browsers were primarily used for displaying static hypertext and related images. Instead of being viewers of static data, current browsers are essentially JavaScript-based programming environments, with support for processing many kinds of data. To provide a seamless user experience, current browsers automatically download and process nearly any data they are requested to fetch, as visualized previously in the threat model in Figure 1. From a security perspective, this is a nightmare, since this combined with the multitude of data formats supported in modern browsers makes the attack surface enormous. The ease of injection of malicious data and homogenous environment makes them a good target for malware, which combined with the evolving standards and growing consumer expectations make securing browsers an unprecedented engineering challenge.

Sandboxing techniques are used to mitigate the effect of vulnerabilities, but this has merely made exploitation more laborious - bypassing the sandbox, address space randomization etc. is still possible, but may require chaining exploits for several bugs together.

Two major open-source browsers, Chromium (used as the basis for Google Chrome) and Firefox are developed in an agile fashion, with major stable releases occurring every six weeks. Web browsers are also one of the most fuzzed pieces of software, as browser vendors have invested heavily in test automation and developed novel instrumentation methods, such as AddressSanitizer. Also, vendor bug bounty programs and the wide install base have attracted third party security researchers. These factors, and the public nature of development make them an ideal case study for the use of fuzzing in test automation.

Browsers quickly gain new features as web standards evolve. The features first appear in nightly builds. The most invasive and risky features are only enabled, for testing purposes, when a special configuration flag is set. As trust increases in the feature (both functionality and robustness), the feature is enabled by default and included in beta and release versions.

With quickly evolving programs, like web browsers, problems occur with the development of fuzzing test case generators. Generation based fuzz testing requires work and time, when new features have to be researched and new generator grammars have to be developed. With mutation based fuzzing the problem is finding sample files, at least when there is a completely new feature and old samples cannot be recycled.

Because Chromium and Firefox are both open-source projects their test suites are publicly available. As new standards are designed, both vendors start to develop new features to conform to the standard. At the same time, they start to add new tests for that feature. Individual tests in these test suites are commonly designed using a vendor specific design model. With a simple conversion script, the individual tests can be converted to standalone files that can be used with a mutation-based fuzzer when testing the new feature. New vulnerabilities and bugs can be revealed from other browsers by samples derived from another vendor's test suite.

Our group participates in these efforts. Since our initial work in this area, finding new vulnerabilities with the same methods has become increasingly more difficult. Increasing the amount of used hardware has not significantly helped. Due to the efforts of both browser vendors and the bug bounty hunter community the "low hanging fruits" have been picked and we have to be smarter with our fuzzing efforts to reach the same results.

In 2010, we used blind mutation based fuzzing, with random samples downloaded from the Internet, and easily found new bugs. In terms of browser features, most of these random files are outdated. In practice this meant that our fuzzing efforts were directed towards the testing of older features in browsers. These features were more likely to be already fuzzed by browser vendor, or the bug-bounty hunter community. To be more effective in our fuzzing, we needed to balance our fuzzing capabilities to cover the whole threat model of the browser more evenly.

As a solution, we decided to use code coverage tools, like SanitizerCoverage, to preprocess our sample corpus. Preprocessing minimizes redundancy from the sample corpus and thus emphasizes samples that stress rarely seen features in the target. Preprocessing also removes samples that use features that are either deprecated, or not yet implemented, in the target. In practice this method has lead to the discovery of bugs in features like Google

Chrome's Content-Security-Policy(CSP). At that time only 0.002% of randomly collected samples included CSP. Also, we found a vulnerability in Mozilla Firefox MP3 support (CVE-2015-0825). At that time, MP3 was marked as an unsupported format in Firefox documentation.

We also use runtime coverage analysis while fuzzing. Runtime code coverage analysis allows our system to automatically collect interesting mutated cases and feed those back to our fuzzing system for further mutating. This process allows our testing system to reach code paths that would have been unreachable with our previous approaches.

We have now found over 110 security vulnerabilities (CVE number) in the stable versions of the major web browsers. In addition, a similar number of vulnerabilities were found in pre-release (unstable, beta) versions, but the vendor does not analyze the security impact as heavily in these cases.

As another approach to add effectiveness to our fuzzing we targeted the fuzzing into individual third-party libraries used in the browser. For example, these libraries include parsers for different media formats. Direct fuzz testing of these libraries allows us to run more test cases, in a given time, than we could run when testing the whole browser. Of course, as a side effect we can miss some vulnerabilities that are not due to an error in the library itself but in the way the browser handles the library. Hence, testing inside the browser is also required. Using runtime code coverage analysis, when fuzzing third-party libraries, allows us to collect a sample corpus that can be later used as a base for browser fuzzing.

## Conclusions

This paper concludes that fuzzing can be integrated into agile software development and test automation, but there are difficulties. The main challenge is that fuzzing needs to continue to show value while requiring minimal effort.

The process used for testing Chromium demonstrates a very advanced way of utilizing fuzzing, which is supplemented by a bug bounty program. Implementing similar infrastructure is beyond most projects, but many activities can be done with limited resources.

New features are added to the threat model, which in the case of web browsers is essentially a list of new supported formats. The features are phased in gradually, and are not enabled until they have been tested for both functionality and security. Finally, comprehensive feature test suites are reused as models for model-based fuzzers or samples for mutation-based ones. The quality is high enough that reasonably high bug bounties can be paid to the vibrant community of external bounty hunters, while being cheaper than in-house testing.

Fuzzing campaigns are not always successful. The greatest challenge is to firmly establish fuzzing in the organization. The best way of doing this is for developers to constantly see new, useful, results. This encourages further investment in testing, and reduces the risk of the fuzzing campaign being a one-man effort that will eventually be abandoned.



The effectiveness of fuzzing depends on how well it is executed. This begins from data flow based threat modelling, which should be continuously updated. Threat modelling should also take results from previous fuzzing into account. If new issues are found with fuzzing, the interface should be a candidate for increased fuzzing efforts. Even limited dumb fuzzing with manual instrumentation can find some low-hanging fruit, and thus should be done for all inputs of the system. The quality of third party components should also be considered.

To stay in use, fuzzing needs to be automated. Even then, the infrastructure will eventually stop finding new bugs. Up to a point, improving test case generation (better models or samples, adding new fuzzers) and instrumentation (use of new type of memory debugger, custom test harness) will continue to find new bugs. Many of these activities only require a small one-time cost, and are easily justified. Others require more significant efforts, including the use of experts.

Coverage guided fuzzing is a promising field, and can improve the quality of fuzz test campaigns. In addition, coverage provides insight into whether placing further effort into improving the test campaign would be useful.

A key question is the availability of the skills required to setup an effective fuzz test campaign. The Software Security Group is pivotal, but eventually testing will have to be done by developers with the proper tools and skills. Setting up ways of working to make fuzzing more effective is one way of doing this. This could be, e.g., standardized ways of building instrumented builds and ways of leveraging results from other activities, like automatically reusing API documentation as a model for fuzz tests.

Some basic skills are required from all development personnel. The fuzzing tools should only require skills that the personnel have. Also, personnel need the skill of understanding crash reports well enough to find the root cause of the bug.

The workflow used by fuzzing is somewhat different from traditional testing and this needs to be accounted for. Instead of being an activity that responds to the previous development increment, automated fuzzing is a background process that needs maintenance. Currently, this requires some expert knowledge, but better tools could also fill this gap.

Coverage-based methods are an efficient method of improving the test case corpus, thus improving test quality. Ideally, they would also make it possible to automatically direct fuzzing efforts into areas, which have recently changed in the code. This would also serve as evidence that the new functionality has been tested.

## **Further reading**

Pietikäinen, P., Kettunen, A., & Röning, J. (2016) Steps Towards Fuzz Testing in Agile Test Automation. *International Journal of Secure Software Engineering*. 7, 1 (January 2016), 38-52.

Sutton, M., Greene, A., & Amini, P. (2007). Fuzzing: Brute force vulnerability discovery Addison-Wesley Professional.

SAFECode. (2012). *Practical security stories and security tasks for agile development environments*. Retrieved Sep. 24, 2015, from

[http://www.safecode.org/publication/SAFECode\\_Agile\\_Dev\\_Security0712.pdf](http://www.safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf)

---



Takanen, A., Demott, J. D., & Miller, C. (2008). Fuzzing for software security testing and quality assurance Artech House.

Zalewski, M. (2015). American fuzzy lop fuzzer. Retrieved April 28, 2017, from <http://lcamtuf.coredump.cx/afl/>

## Product security incident response team

Raino Lintulampi, Bittium Wireless.

### *Executive summary*

This article introduces one approach how the security of a product's software can be monitored during the product's life cycle. Product Security Incident Response Team (PSIRT) is an organization whose responsibility is to proactively scan new vulnerabilities related to the product software and react if those can be found. Vulnerability scanning is an important part of PSIRT operation. A concrete example is presented how the vulnerability scanning can be organized using a local copy of public CVE data bases and advanced reporting tool.

### *Introduction*

All organizations are connected to the outside world using computer networks. Companies and societies are depending on working communication networks. The security of the communication is more important today than ever before. Several means are applied to improve the security. Computers have anti-virus and malware detection software. Firewalls protect companies' networks; all connections are passed through proxies which makes it easier to leverage organization level policies. Network traffic can be monitored with intrusion detection system which alert if suspicious traffic is detected. There are administrative tools that help information management personnel to check the status of computer software versions and automatically install security patches.

New and even bigger challenge comes when different control systems including critical infrastructure and autonomous machines are connected to large scale networks which can be accessible from public internet and therefore being vulnerable for same kind of threats as computer networks. An industrial site can have hundreds, even thousands of different types of devices connected to industry network. Although accessing these devices is much harder than computers in a computer network, it is still possible. Same kind of administrative tasks are much harder to execute for industrial network than in typical computer network which includes Windows machines and Linux servers, for example. Therefore it is crucial that all the vendors follow good practices for building secure devices from the beginning. However, when a product is integrated from hundreds of open source components, it is impossible to test all the software components thoroughly. Therefore it is important that the known and newly reported vulnerabilities for the software components are constantly scanned.

### *Definitions and benefits of having PSIRT organization*

Product Security Incident Response Team (PSIRT) function resembles some of the Computer Security Incident Response Team (CSIRT) functions. European Network and information Security Agency (ENISA) has published several reports and guides how to organize computer network security management. One is "A Step-by-step Approach on How to Set-up CSIRT" which is followed and adapted in this article to PSIRT function.

One of the main responsibilities of CSIRT is to monitor the computers and machines installed and used within the organization. CSIRT sees the software as black boxes, for example Windows 7 SP2, Ubuntu Linux 14.04.2 or Firefox browser 50.01. CSIRT actively monitors vulnerabilities reported to these system and applications and installs patches as soon as vendors are providing them. PSIRT on the other hand is the organization of product (embedded system, software application) vendor who monitors and solves security issues found in the product. PSIRT works closely with the product's maintenance and after sales team who are responsible for finding the technical solutions.

A PSIRT is a team of experts who provides information of possible product security vulnerabilities and/or threats to a product development team and a product maintenance team. A PSIRT team may cooperate also with other teams in order to mitigate security risks. One possible team to cooperate is Open Source Governance (OSG) team. When an open source component is planned to be used in the product the open source licenses are analyzed so that there are no licensing and commercial issues when using the component. At this phase OSG can ask PSIRT to check that there are no security issues in the open source component. If the licensing and security check are passed then the open source component can be used in the product.

PSIRT has only internal customers when CSIRT may act internationally or nationally, and may sell services to anyone asking of them. PSIRT focus is on company products and that organization has the best knowledge of the products, or should be.

While CSIRT activity within information management team is a common task nowadays, a separate PSIRT function is not necessarily that common at least as an independent function. There are, however, several benefits to separate security issue handling from the product development and maintenance:

- Each product team does not need to invent the new process for handling security issues. All product teams benefit from the process improvements at the same time.
- There is one point of contact.
- Cooperation with other teams, for example IM CSIRT function and OSG, is easier.
- Search from common vulnerability and exploitation database requires exact search terms. With wrong search, vulnerabilities may not be found or the number of false positive findings increases which require extra work.

### ***Possible PSIRT services***

The actual services depend on the organization, the number of products and many other issues. Table 1 lists some of the services which PSIRT could provide. The same categorization is used as used for CSIRT in ENISA's guide. Note that for example Vulnerability analysis is a reactive service in CSIRT as in PSIRT it is a proactive service.

**Table 1 PSIRT services**

PSIRT			Product development and maintenance
Reactive services	Proactive services	Quality and education	
Incident handling	Vulnerability analysis	Awareness building	Incident analysis
	Alerts and warnings	Education/training	Vulnerability handling
	Development of security tools		Vulnerability response
	Technology watch		

There is a shift from Reactive Services to Proactive Services in PSIRT services compared to CSIRT. This is natural because the PSIRT function is part of vendor's operation and it focuses on products and not on monitoring computer network. PSIRT function can also be seen as part of product's quality assurance.

The only reactive service is incident handling. Incident means that either a customer or for example a security organization has found a vulnerability in the product which has been or can be exploited. Incident handling must be started immediately – in PSIRT operation that means the next working day earliest – and before that the relevant information must be gathered from the customer. The customer's CSIRT organization is responsible for the immediate damage control. They may isolate sections of the network, filter the network traffic or shutdown the devices which are affected or threatened by intruder or malware activity. CSIRT is also responsible for gathering the artifact information and possible forensic evidence. Artifact information is delivered to the product vendor. Depending on the products, the malware, virus or threat may be targeted directly to the product or it may have been affected due to malfunction in other device in the network, for example a firewall or a server. This information is needed when an incident analysis is done. The analysis includes the verification of suspected vulnerabilities and the technical examination of the hardware or software vulnerability to determine where it is located and how it was or can be exploited. Because the analysis most likely includes reviewing source code and even debugging the code, it can be done only by people familiar with the product. Before the vulnerability can be verified the incident has to be reproduced on a test system. It affects prioritization and response dead line. If the root cause is not in the product then corrective actions can be postponed to the next scheduled software update. But if the product is critical part of the network then the response must be done as soon as possible. The response may include patches, fixes, and workarounds. If the vulnerability has been exploited and it is in a critical device then it is important to notify other customers as well. The main point of PSIRT function is to avoid this kind of situation; however, incidents can happen because no process is perfect. Therefore the open communication is crucial between the PSIRT and CSIRT because it shows that the vendor's quality assurance in general and specifically the PSIRT functions have not succeeded.

The most important proactive service is vulnerability analysis. That means that the vulnerabilities of the components in the product are scanned. This involves two parts. The first part is doing security testing of component interfaces. Fuzzy testing which was

described in Chapter *Utilizing Fuzz Testing in Agile Test Automation* is a powerful tool for that. Security testing is an R&D phase operation and it is not executed any more after the product release.

Because a product may include hundreds of (open source) components, only critical interface components can be thoroughly tested. Also no test can find all the code flaws or vulnerabilities. Therefore known and newly reported vulnerabilities for the components must be constantly scanned. That is an operation which has to be carried out from R&D phase until to the product's end of life. Chapter *Setting up product vulnerability checking* describes one solution how vulnerabilities can be scanned. Vulnerabilities found in normal product vulnerability scanning are further handled, analyzed and if needed corrected by the product development or maintenance organization.

PSIRT can be a source for vulnerability report to CSIRT of its customer organization. One should also notice the difference in focus. CSIRT sees the product as black box, PSIRT sees the product as white box and it checks vulnerabilities of the software components from which the product is built, therefore, PSIRT may check vulnerabilities for thousands of items for one product.

PSIRT may also act as a coordinator for developing, evaluating and maintaining security related tools which are commonly used in an organization. Vulnerability scanning tool described is an obvious example but also fuzzy testing tools and static scanners may be evaluated by PSIRT.

PSIRT function can be seen as part of the quality assurance process. Training and awareness building can be defined also as a goal for PSIRT.

There is also a difference in how CSIRT and PSIRT should respond to the found incident or vulnerability. Because PSIRT is responsible for securing the product, the immediate actions are on the responsibility of the CSIRT. Furthermore, the corrective actions are not like shutting down of network sections or a computer which has a malware, but the product software itself has to be changed. Therefore if organization's CSIRT is 24/7 activity, PSIRT is the office hour activity because it is not likely that code debugging can be started e.g. on Saturday night. On the contrary, because the corrective action is the patch for the product software, then it has to be done and tested thoroughly. However, it could be reasonable to describe the process in Service Level Agreement how the corrective actions are taken. For example, the analysis of the product component vulnerability is started within two working days. Confirmed vulnerabilities are corrected within 30 working days and patches are delivered to customers within 5 working days after the patch is approved. Depending on whether the vulnerability was detected in normal proactive process or was it reported as an incident, the processing times could vary. Depending on the product and number of installation base several patches may be combined together.

Communications towards customers differ as described in Table 2. It is not necessary to inform consumers about the vulnerabilities which have not been corrected. That information would go public and if the vulnerability has not been exploited yet, after that it would be definitely exploited by hackers. Therefore security updates are reasonable way to

communicate with consumers. Devices contact (when allowed) to the vendor's software update servers and download new software versions there.

In business to business relationship the situation is different. The more critical the product is the more important is the open communication. The customer may even require that any detected vulnerability is informed to them at the time it is detected. Anyway it could be a good idea to open the process also to the customers so they would understand how their vendor handles security issues. That increases trust and may even be competitive advantage.

**Table 2 Communication options to different customer types**

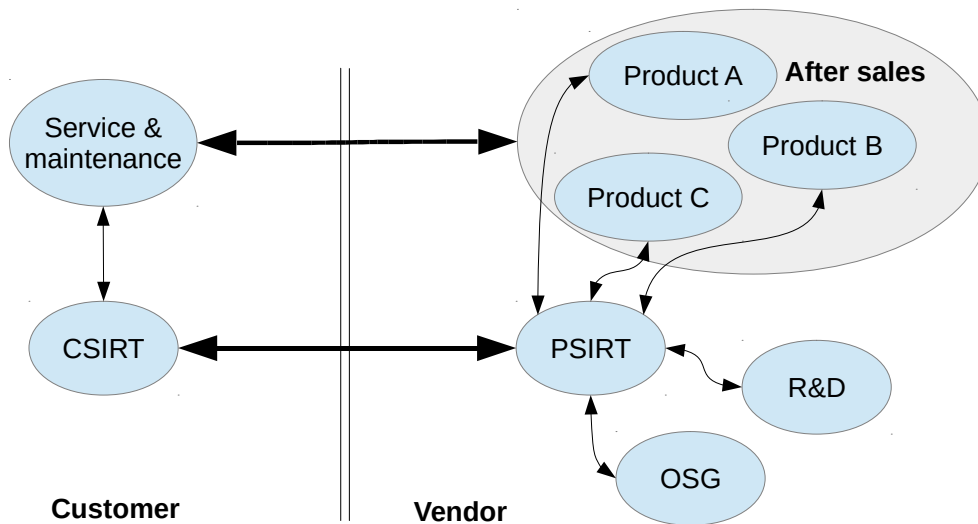
Customer type	Communication means
Consumer	Security patches
B2B Customers	PSIRT process description Service agreement including patches
Security critical customer	PSIRT process description Vulnerability reports Service agreement including patches

However, when devices are part of critical infrastructure, they cannot communicate via internet with any update servers. The vendor has to agree in Service Level Agreement how security patches are delivered and how they can be updated if needed. With a typical B2B customer that is handled between the vendor after sales / product maintenance organization and the customer's service organization and PSIRT is not involved in the software update process. Although the vendor has corrected some security vulnerabilities and delivered them, the customer may not want to install the patch because of 'do not correct it if it is working' attitude. That may be a reasonable decision. If the customer, however, requires vulnerability report when that is detected, then the communication happens between the PSIRT and the customer CSIRT organizations as described in Figure 10. It has to be agreed clearly how the communication is done and how it is secured. No security information can be sent in clear text. PSIRT may also have to report how the vulnerability has been processed in the vendor's organization. This can be done in a same way as vulnerability reporting or alternatively CSIRT may have a limited view to PSIRT's ticket processing system.

## PSIRT organization

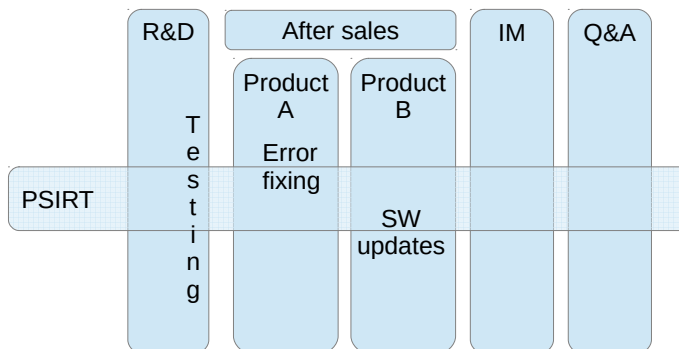
There is not only one way to organize PSIRT activity but it depends on several issues. One thing is, however, common to all solutions. The organization has to define what PSIRT does. PSIRT operation creates costs. Costs may come from new tools and equipment and from personnel cost. How small the operation is still someone needs to do something and that needs money. However, the overall cost may be less than with the current operation but usually organization sees all new functions as an additional cost. Therefore there has to be clear mission statement what the new function – PSIRT – does and delivers. Here is an example statement:

*PSIRT ensures that our customers have secure products until the product's end of life.*



**Figure 10 Communication between the product vendor and the customer**

The above statement sounds quite good. However, it is also quite wide. The product life cycle includes the product development and maintenance. Is it PSIRT's responsibility to make security testing of the product during an R&D phase? Are security related errors different than other errors? If the mission statement is taken literally, these are the questions which should be answered. The mission statement leads easily to a horizontal PSIRT organization as described in Figure 11. This may be too complex structure to start PSIRT activity. A simpler approach may be a better starting point.



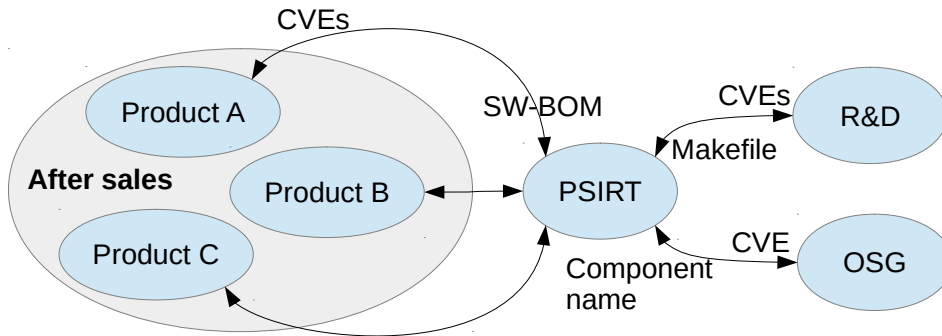
**Figure 11 A horizontal PSIRT organization**

An example of a simpler mission is:

*PSIRT provides information and training about security vulnerabilities to product teams.*

This statement is concrete but at the same time expandable in a way that it allows adding new functions to PSIRT based on the experiences. The starting point of this kind PSIRT could be an activity that first focuses on providing product related vulnerability information to products' maintenance team and administrating the vulnerability scanning tool as depicted in

Figure 12. PSIRT is just an information sharing entity. What happens to the vulnerability reported by PSIRT is out of the scope of PSIRT's responsibility. Communication and delivering the patches to customers is not visible to PSIRT.



**Figure 12 Information sharing PSIRT**

Later PSIRT may also evaluate and maintain security testing tools used at R&D phase including training. Next step could be to establish a communication channel between PSIRT and customer's CSIRT. That case was elaborated in Figure 12. Eventually the PSIRT organization may become very close to the one presented in Figure 11. It is questionable whether product maintenance activities should be included in PSIRT because the patching the product software requires always a deep understanding of the code. Good communication is needed between the teams to ensure that the product's security is kept constantly at the highest level.

In next chapter it is described how the product software vulnerability scanning can be done automatically.

## Setting up product vulnerability checking

### Introduction

Checking of software vulnerabilities is an important part of software development and maintenance. It is a fundamental part of PSIRT operation as described in the previous chapter. Vulnerability scanning is not a one shot activity but it has to be done on regular basis because new vulnerabilities may be found at any time. At a development phase it is important to check that a new software component, which is planned to be used, does not contain such vulnerabilities which prevent the use of the component in the product. In case of an open source component the initial vulnerability analysis may be done as part of open source governance when also open source licenses are checked. After that the software components should be checked constantly.

### CVE-search

For vulnerability scanning the access to the vulnerability database is mandatory. There are several public Common Vulnerability and Exposure (CVE) databases like <http://cve.mitre.org>, <https://web.nvd.nist.gov/view/vuln/search>, <https://www.exploit-db.com/> and <https://www.circl.lu/services/cve-search/>. There are also product specific databases as WindRiver's <https://www.windriver.com/security/cve/main.php> and non-public commercial database as Risk Based Security's <https://www.riskbasedsecurity.com/vulndb/>. Because security is everyone's concern, the basic CVE databases are public and common for all

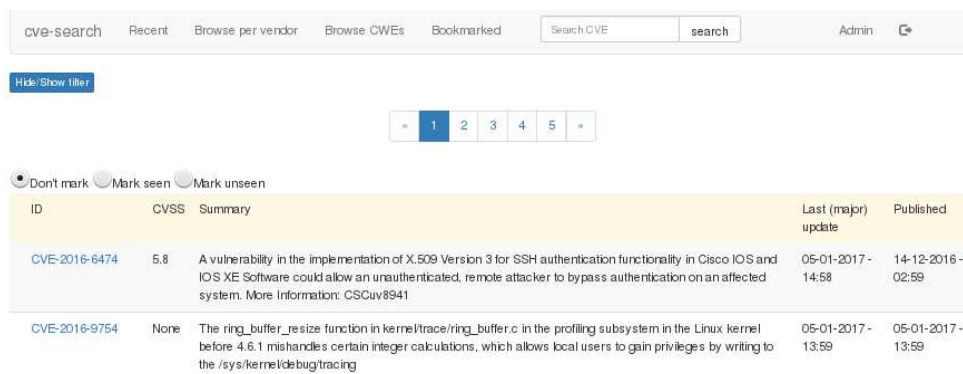


database providers. However, there are slight differences how data is presented. For example Mitre and NVD has a very close relation. Mitre feeds the CVE list to the [U.S. National Vulnerability Database \(NVD\)](#), which then provides upon the information included in CVE entries enhanced information for each CVE Identifier such as fix information, severity scores, and impact ratings. NVD also provides advanced searching features such as search by individual CVE ID; by OS; by vendor name, product name, and/or version number; and by vulnerability type, severity, related exploit range, and impact.

Although public CVE databases can be accessed directly, it may expose critical information if the access is used to query information related for the product software. If the queries are monitored then a listener can get the information about the software components used in the product and possible uncovered vulnerabilities which can be utilized on targeted attacks. The better approach is to setup and maintain a local CVE database from which the product specific CVE searches are done.

*Cve-search* (<https://github.com/cve-search/cve-search>) is a tool to copy CVE information from multiple sources and perform local searches for known vulnerabilities. *Cve-search* is copyrighted by Wim Remes, Alexandre Dulaunoy and Pieter-Jan Moreels and is available under BSD-3-Clause license. The tool uses MongoDB to which data from different CVE and security databases is copied. Database handling and search tools are implemented in Python3. The main source for CVE data is NIST NVD database but information is also queried from Exploitation reference from D2 Elliot Web Exploitation Framework ([www.d2sec.com](http://www.d2sec.com)) and [MITRE Reference Key/Maps](#) and Offensive Security's sponsored Exploit Database.

*Cve-search* has a minimal web interface for searches as depicted in Figure 13 and Figure 14, but it does not offer as versatile features as for example NVD search interface. However, the main functionality of *Cve-search* is not the web interface but possibility to update the local database regularly and make easily command line searches from it which can be executed from scripts and then post-processed with different tools.



ID	CVSS	Summary	Last (major) update	Published
<a href="#">CVE-2016-6474</a>	5.8	A vulnerability in the implementation of X.509 Version 3 for SSH authentication functionality in Cisco IOS and IOS XE Software could allow an unauthenticated, remote attacker to bypass authentication on an affected system. More Information: CSCuv8941	05-01-2017 - 14:58	14-12-2016 - 02:59
<a href="#">CVE-2016-9754</a>	None	The ring_buffer_resize function in kernel/trace/ring_buffer.c in the profiling subsystem in the Linux kernel before 4.6.1 mishandles certain integer calculations, which allows local users to gain privileges by writing to the /sys/kernel/debug/tracing	05-01-2017 - 13:59	05-01-2017 - 13:59

**Figure 13** *Cve-search* main web view

cve-search

Recent

Browse per vendor

Browse CWEs

Bookmarked

Search CVE

search

Admin

CVE-Search / CVE-2015-0293

ID

CVE-2015-0293 ☆

Copy to Clipboard

Summary

The SSLv2 implementation in OpenSSL before 0.9.8zf, 1.0.0 before 1.0.0r, 1.0.1 before 1.0.1m, and 1.0.2 before 1.0.2a allows remote attackers to cause a denial of service (s2\_lib.c assertion failure and daemon exit) via a crafted CLIENT-MASTER-KEY message.

References

- <http://kb.juniper.net/InfoCenter/index?page=content&id=JSA10680>
- <http://lists.apple.com/archives/security-announce/2015/Jun/msg00002.html>
- <http://lists.fedoraproject.org/pipermail/package-announce/2015-March/152733.html>
- <http://lists.fedoraproject.org/pipermail/package-announce/2015-March/152734.html>
- <http://lists.fedoraproject.org/pipermail/package-announce/2015-March/152844.html>
- <http://lists.fedoraproject.org/pipermail/package-announce/2015-March/152899.html>

Vulnerable Configurations

- OpenSSL Project OpenSSL 0.9.8ze Change Log Vendor
- OpenSSL Project OpenSSL 1.0.0
- OpenSSL Project OpenSSL 1.0.0a
- OpenSSL Project OpenSSL 1.0.0b
- OpenSSL Project OpenSSL 1.0.0c
- OpenSSL Project OpenSSL 1.0.0d

CVSS

Base:

5.0 (as of 20-03-2015 - 16:00)

Impact:

2.9

Exploitability:

10.0

CWE

CWE-20

CAPEC

- [Buffer Overflow via Environment Variables](#)
- [Server Side Include \(SSI\) Injection](#)
- [Cross-Zone Scripting](#)
- [Cross-Site Scripting through Log Files](#)
- [Command Line Execution through SQL Injection](#)
- [Object Definition Mismatch Injection](#)

Access

Vector	Complexity	Authentication
NETWORK	LOW	NONE

Impact

Confidentiality	Integrity	Availability
NONE	NONE	PARTIAL

Fedora

fedora

FEDORA-2015-6951

**Figure 14 CVE details shown, partial view**

An example of command line search is described below with clear text output.

```
$ search.py -p openssl:openssl:1.0.2h
CVE      : CVE-2016-2177
DATE     : 2016-06-19T21:59:02.087-04:00
CVSS     : 7.5
OpenSSL through 1.0.2h incorrectly uses pointer arithmetic for heap-
buffer boundary checks, which might allow remote attackers to cause
a denial of service (integer overflow and application crash) or
possibly have unspecified other impact by leveraging unexpected
malloc behavior, related to s3_srvr.c, ssl_sess.c, and tl_lib.c.

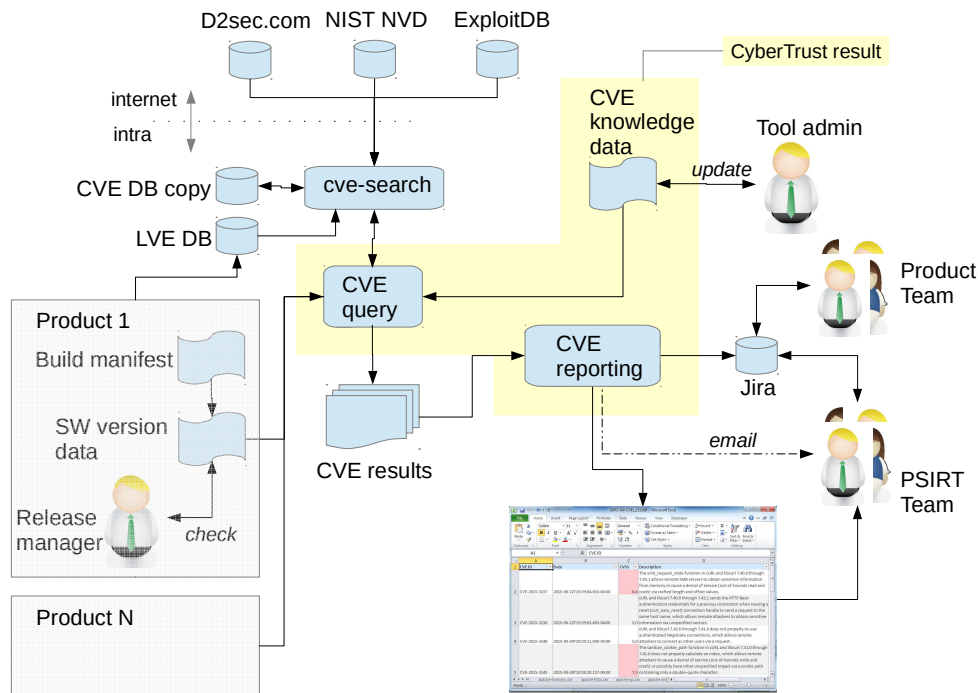
References:
-----
.cut.
cpe:2.3:a:openssl:openssl:1.0.2h
```

Output of the search can be stored in text, xml, json or html format. The content of the specific CVE ID can be searched as well. Also free text searches are possible.

The next chapters describe how *Cve-search* features can be utilized in product vulnerability tracking process and what should be considered in such a process.

## CVE database

Figure 15 depicts how vulnerability scanning can be built as part of PSIRT function. PSIRT function in this scenario is the same as described in Figure 12 and PSIRT just provides CVE information to its local customers.



**Figure 15 Vulnerability scanning framework**

The key point is the local copy of public CVE databases to which CVE data is collected from public sources using *Cve-search* program and from which local queries can be executed. CVE DB copy should be updated every day. Typically the number of new vulnerabilities added to the data base is from few tens to couple of hundreds each day. LVE DB (Local Vulnerability and Exposure data base) is a special data base used only by a company itself. If open source components are tested for example using fuzzy testing tools, new and not publicly known vulnerabilities may be found. It is a good practice to report findings to the upstream project; however that is not always possible for example if the project is not active any more. Also there may be a significant delay before the reported vulnerability has been corrected and reported to the public CVE data bases. Therefore the error reports can be stored to the special LVE data base in a format which *Cve-search* can read. Now all locally reported security issues are copied to the CVE data base copy and they are visible in future CVE search for the component. LVE data base is important for the large organizations with number of products because it is very difficult to know who is using the component and hence who should be reported about the findings.

A product related CVE queries are executed from the script file which is automatically generated using a product's software information and CVE knowledge data. Software information is described in a XML file which contains a component name and version, see Figure 16.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SWversions>
  <SWcomponent>
    <Name>kernel</Name>
    <Version>3.4.0</Version>
  </SWcomponent>
  <SWcomponent>
    <Name>btconfig</Name>
    <Version>2.10</Version>
  </SWcomponent>
  <SWcomponent>
    <Name>busybox</Name>
    <Version>1.22.1</Version>
  </SWcomponent>
</SWversions>
```

**Figure 16 Product software's version information**

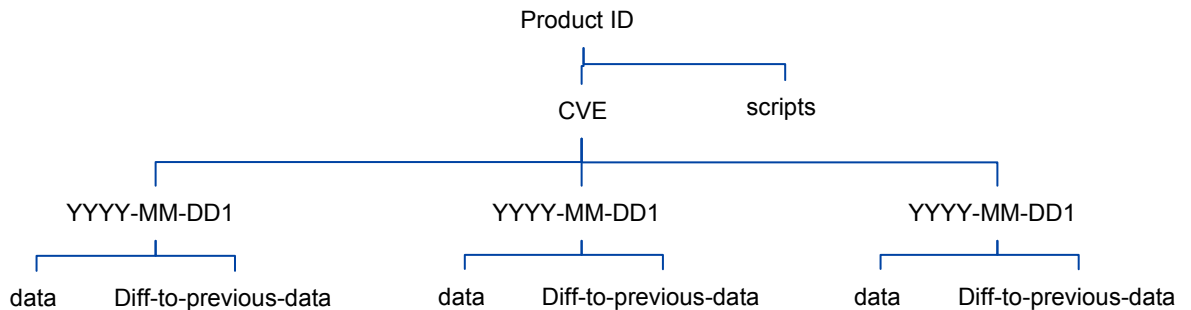
A second input for the CVE script generator is the CVE knowledge data. It is not always obvious how the software component is named in the CVE data base and therefore what is the correct query term. If a wrong search term is used then critical vulnerability information may be missed or the search gives a large number of false positive results which causes unnecessary extra work. CVE knowledge data includes the information for the query and how the version numbers should be interpreted, see Figure 17. Digits field defines how many digits of the component version number are used in the CVE query. Some cases patch level information – digit 3 – is not used in reported vulnerabilities and if 3 digits version number is used in a search no vulnerabilities are found which may miss again the critical information. When a new component is added to a product then the component query terms are manually checked and the CVE knowledge data is updated if needed.

```
<SWcomponent>
  <Name>curl</Name>
  <CVELink>haxx:curl</CVELink>
  <Digits>2</Digits>
  <Status>Checked</Status>
</SWcomponent>
<SWcomponent>
  <Name>expat</Name>
  <CVELink>libexpat:expat</CVELink>
  <Status>Checked</Status>
</SWcomponent>
```

**Figure 17 Example of CVE knowledge data**

Execution of the generated CVE query script can be scheduled for example using *cron* on a wanted interval. Depending how critical the product is, the query can be done daily, in not critical product case the CVE query can be done only weekly basis. Because the queries are fully automatic there is no harm to do that on daily basis. On the contrary, then processing of a possible new critical vulnerability can be started without a delay. The CVE query script fetches all the product related CVE information from the CVE data base copy. The structure of the fetched CVE data is presented in Figure 18. The data consists of two parts; data describes all the vulnerabilities found in the data base, diff-to-previous-data describes only the new or updated vulnerabilities. The latter information is generated by the CVE reporting tool that is activated after CVE query which fills the data information.

---



**Figure 18 CVE results structure**

CVE reporting tool checks if there are new CVEs or if earlier CVEs have been updated. Only those CVEs that are above a predefined CVSS criterion are reported and either a Jira issue is generated or an email is sent or both. Depending on the organization and product structures the CVE reports can be sent to PSIRT team or directly to a product maintenance team. Jira or similar tools like Bugzilla are a preferable solution for reporting because they make it easier to keep a track how vulnerability has been handled. If needed an Excel report can be generated from the detected CVE results.

CVE reporting has few configuration options as described in Figure 19. Mails section defines to whom the indication of CVE findings is sent. It is possible to generate automatically a Jira issue about the finding. In that case a special email address needs to be defined in Jira which is added to `<to/>` field. In this case a person in `<cc/>` field is automatically assigned as a responsible person for the issue. `<Product/>` field defines the product for which the found CVEs are related. `Product_name` is added to email's subject field. `<cvss/>` field defines the criteria for indicating the CVE finding. Only findings whose CVSS are equal or greater than value in `<cvss/>` field are reported. CVSS can have a numeric value between 0 and 10.

CVE server name is the first choice to find out the database location. If it is omitted then IP address is used. If debug option is enabled then mails are generated but they are not sent but printed to a console. That is a good option for testing before spamming a product team with possible hundreds of mails if settings are incorrect.

```

<configurations>
  <product>Product_name</product>
  <cvss>5</cvss>
  <mails>
    <to>
      <mail>rock.bar@company.com</mail>
      <mail>blues.bar@company.com</mail>
    </to>
    <cc>
      <mail>jazz.bar@company.com</mail>
    </cc>
    <bcc>
  
```

```
<mail>foo.bar@company.com</mail>
</bcc>
</mails>
<cveServer>
  <name>cve.intra.company.com</name>
  <address>192.168.1.10</address>
</cveServer>
<debug enabled='yes'>john.developer@company.com</debug>
</configurations>
```

**Figure 19 CVE reporting configuration**

## Conclusions

Product security is not something which is built into the product during the development and then forgotten. Especially today when the software is built from hundreds of open source components it is impossible to verify thoroughly all the components. Therefore the vulnerabilities should be monitored constantly. That is an important function of Product Security Incident Response Team (PSIRT) which was introduced in this article. There are no strict guidelines how the organization should organize PSIRT. It depends on the number of products, company size and many other things. A solution for vulnerability scanning was introduced. The presented framework was developed in the Cyber Trust program and it has been successfully used in Bittium. However, not all open source projects report formally the security flaws found in the project as CVEs. Many security errors are corrected and the track for that can be found only in the changelog or version control system's commit comments. That problem was studied at The University of Oulu but further research is still needed.

## Further reading

A step-by-step approach *on how to* set up a CSIRT, Deliverable WP2006/5.1(CERT-D1/D2), ENISA. (<https://www.enisa.europa.eu/publications/csirt-setting-up-guide>)  
Moir West Brown, Don Stikvoort, Klaus-Peter Kossakowski, Georgia Killcrece, Robin Ruefle, Mark Zajicek, Handbook for Computer Security Incident Response Teams (CSIRTs), (<http://www.cert.org/archive/pdf/csirt-handbook.pdf>)

## Device Management by 2020

Tero Takalo, Capricode

### *Executive Summary*

Nowadays MDM technology is dominated by main mobile platform/OS providers Apple and Google who have strong guidance of what can and what cannot be done by MDM. Also few large players such as VMware (with Airwatch), MobileIron, IBM (with Fiberlink) and Microsoft have dominant market positions leaving room only for SME target markets. This means that in future we see room for specialization in MDM (e.g. secure Android manufacturers for special markets) and on the other hand large growth potential in IoT markets to make growth of MDM players possible if they are willing to make customization and have flexibility to adapt to customer needs. The main market in IoT is not in enterprise usage as in traditional MDM but in OEM device manufacturer needs which mean that needs for functionality, scalability and cyber security are on totally different level than in traditional MDM.

### *Introduction*

MDM has become a commodity rather than a special system. Large players like Microsoft (Intune) and IBM (MaaS360) offer some scale of device management functionalities as side products in their other offering making it difficult to other MDM system provider to get into their existing customers.

OS fragmentation is no longer an issue as Android and iOS dominate in markets. Instead the challenge comes from Android version and OEM (original equipment manufacturer) fragmentation and different implementations of the features or support for different features.

In addition to OEM fragmentation in general, many OEM's have made additions to basic OS MDM functionalities. Samsung (Knox) and HTC have been the most active OEMs. Supporting vendor specific additions vary a lot among MDM vendors.

Compared to earlier needs for MDM for setup, backup and application provisioning, the main need nowadays is in increasing security rather than enhancing employee satisfaction. This is due to evolution of mobile platforms which provide standard solutions for example backup, and user friendliness in the setup of e.g. email where the user no longer needs to have technical expertise. Instead of MDM device side functionalities the evolution and differentiation of the MDM solution is related to wider Enterprise Mobility Management (EMM) aspects in the backend; for large enterprises there is need for other system integration, application management, content management, identity management and master data management. However, the need for these functionalities depend highly on the enterprise size and IT systems and IT management technology level.

EMM is mainly a systematic IT process for defining tool chain and policies required for properly manage and secure mobile device fleets of the company. The evolution BYOD (Bring Your Own Device) instead of (limited) variety of company provided devices has brought a new aspect to MDM. Despite of the need for MDM solutions the predicted high

growth presented by Gartner and other market research companies for MDM has never been a reality. The market growth has been rather slow evolution than a large revolution.

## ***Mobile Device Management future - from mobile devices to IoT***

Digitalization and cyber security trends will continue and form basis also for the device management needs and evolution. Everything will be connected, all data is analyzed and cyber security awareness brings market growth possibilities for device management products. Product life cycle and window of opportunity for growth is all the time shorter and all markets are global.

In emergence of IoT one important issue to consider is who will take the responsibility for IoT devices. In case of company using IoT as part of their process (e.g. factory) the answer is quite obvious that IT department, that manages also all other devices, takes responsibility over IoT as well. In a case where the IoT enables new business via digitalization the business unit could take responsibility directly. Usually this is done utilizing a separate innovation department in the research and development organization or Operational Technology (OT) organization.

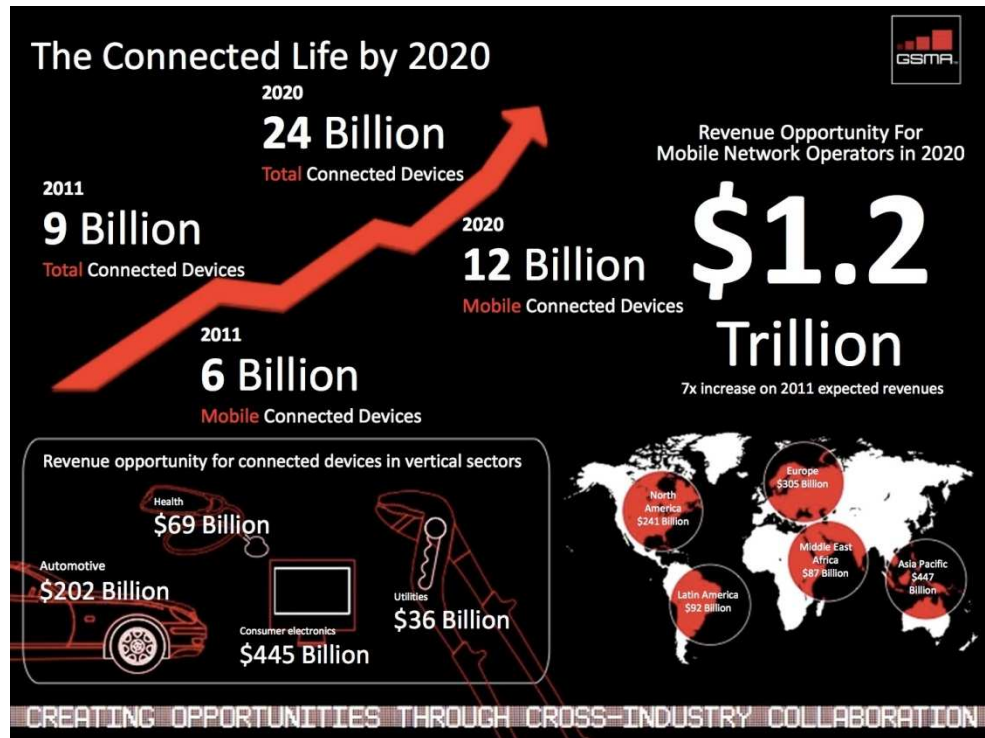
In the past, Information Technology (IT) and Operational Technology (OT) were seen as two distinct domains of a business. The former focused on all technologies that were necessary to manage the processing of information, whereas the latter supported the devices, sensors and software that were necessary for physical value creation and manufacturing processes. One of the factors that is reshaping IoT market is the convergence of Information Technology and Operational Technology which is basically a “must have” in order to scale to IoT vision device amounts and to keep security at proper level.

Convergence of networks - both industrial (OT) and enterprise (IT) are enabling applications such as video surveillance, smart meters, asset/package tracking, fleet management, digital health monitors and a host of other next-generation connected services.

In case of equipment manufacturer (OEM) to make IoT enabled devices (e.g. RFID readers or mining machines) the answer to responsible organization is even more obvious and business unit OT and/or innovation department must be in control of the R&D of management environment and other tools needed for keeping device fleet updated and secure.

The number of IoT devices will grow extremely fast, although estimations for speed of that growth vary a lot. Figure 20 illustrates GSMA organization estimation for cellular network connected device number:





**Figure 20 Estimation of growth of connected devices**

Emergence of IoT brings a few new challenges:

1. Scalability: need to handle much larger number of devices in the system.  
Example: an organization of 1000 employees needs about 1000 devices to MDM system for handling employee smart phones. But the same organization might be handling fleet of 10 million remote readable electricity meters, meaning that IoT device management systems must be able to scale to 10 million devices compared to 1000 of MDM.
2. No human interaction. All actions and processes in a device starts from an initial discovery and enrollment needs to happen without a human interaction as the IoT devices generally do not contain any user interface and on the other hand the scalability cannot be realized if a human action is needed in the process for each device.
3. Standards: Standardization activities are too slow to properly address the challenges of IoT expansion. This has resulted in forming of different (and competing) alliances in the connectivity, protocols and other technical items in device-to-device and device-to-cloud communication. Here are few examples of different alliances:
  - [Industrial Internet Consortium](#)
  - [Open Interconnect Consortium](#)

- [AllSeen Alliance](#)
  - [Thread Group](#)
  - [IPSO Alliance](#)
  - [IoT Eclipse](#)
  - [OneM2M](#)
  - [Bluetooth SIG](#)
  - [Internet of Things Consortium](#)
  - [LoRa Alliance](#)
4. IoT platforms: Market analysts have counted over 700 commercial IoT platforms which mean that the whole landscape of the IoT platforms is very scattered. From device management perspective the DM system needs to be able to integrate well to the platform and bring extended / advanced functionality, automation and other aspects over the standard device management functionalities of the platform.

### ***From commodization to specialization***

Current general MDM and other enterprise systems will continue in that role. In addition to those there will be emergence of niche markets for specialized / customized MDM solutions for special organizations product purposes (e.g. security organizations, governmental organizations, army, high security enterprises) that need tighter management / control features than available by general solutions.

Second track in this specialization is emergence of secure smart phone manufacturers (such as Bittium, Gryphon, Airbus, etc.). They provide whole solutions usually built on tweaked Android OS and that gives possibilities to make advanced features also in MDM systems.

Third track in new markets and customization needs is IoT systems. Since IoT in general is still missing standard interfaces, protocols, etc. for making generic device management systems possible, it will be still many years from now that any and all IoT device management will need some level of custom adaptation in device level and custom integrations in the backend systems.

### ***Technical requirement trends***

The specialization / customization will bring new technical requirements. Some of the requirements are related to device / backend functionalities but there are also many requirements for protocols, networking and OS support as well.

New MDM functionalities needed in near future, especially in high security use cases, are for example:

- Advanced monitoring / diagnostics
- Fetching / parsing log files from devices
- Firmware updates
- Application revocation
- Application and process whitelisting
- Stronger authentication / identification of devices
- Stronger overall control over devices
- Advanced admin user rights control
- Advanced automation in device enrollment

A list of needed new technologies, especially in IoT, are for example:

- New communication methods support such as Sigfox, LoRa, Wirepas, Kalliot
- New protocols support such as MQTT, NGTP, SNMP, CoAP, LWM2M
- Multihop (connect via mobile phone Bluetooth to sensor, etc.)

Major technological development needs to be done in scalability of MDM systems in order to address vast IoT device amounts (environments that consist of millions of managed devices). Earlier MDM systems were not meant for real time operations, transferring large data files (e.g. firmware updates) or in general able to scale up to millions of devices either due to limits in backend scalability or lack of automation.

## **Cyber Security**

In cyber security the main question is: who is responsible of the security? Is it company's IT department, end user, device manufacturer or some public organization? Or can responsibility be "outsourced" to 3<sup>rd</sup> parties? Responsibility question is not limited only to organizations but should be considered also from private person/property perspective as well. Consensus in responsibility questions must be found and connect that to technical practicalities and tool chain, many of those have link to device management system requirements.

A study by HP in 2014[1] showed that 70 percent of IoT devices contain serious vulnerabilities. There is undeniable evidence that the dependence on interconnected technology is defeating the ability to secure it. The situation has become even worse since the study because due to expansion of installed devices the number of vulnerable devices increases even in case when the percentage of vulnerable devices in new devices would be lower than in the study. It is expected that number of exploits and vulnerabilities will continue rising as it has been during past few years. Especially as IoT is still so early in its evolution and legacy M2M systems, routers, etc. internet connected devices having strong legacy of neglecting security by default there is huge amount of work to be done before reaching even some satisfying level of security.

Making devices secure and safe is one of most important needs for device management systems. Just by ensuring correct settings and setup, and keeping firmware updated improves cyber security to quite good level even without any advanced threat protection or monitoring. However, this is true only against mass exploits and botnet attacks that target to manufacturer default settings or known vulnerabilities. Mirai botnet is a good (or bad) example as it has been able to collect the largest botnet ever just by targeting to devices which have a simple protection (e.g. "root / root" credentials) and management console or root access ports available to internet.

Life cycle management of devices is one very large problem both in mobile and IoT domains. Device manufacturers are willing to make updates to their devices only for a short period after reaching end of production. This support period is a lot shorter than average lifetime of devices, making devices vulnerable for a long period of time before their use has deceased. Another issue in security updates is that even if OEM produces updates, in many cases they are not updated to devices as the clear responsibility of security is missing in organizations.

A practical example of life cycle management is the known [heartbleed](#) vulnerability in the OpenSSL library. Most of the older Android mobile phones only support OpenSSL versions which have the vulnerability and the corrective update is not available from OEM's. Still these devices can be widely used and organizations want to keep them in MDM system. However, Android platform MDM client available in Google Play may not support any more those devices because the MDM client should also include the vulnerable OpenSSL library and such applications have been rejected by Google.

Zero day vulnerabilities are another issue to be considered. It is not possible to directly protect against this kind of "unknown" threats but instead the monitoring systems and algorithms should be at such a good level that they bring preventative protection and detection possible also against zero day exploits as well as possible. Currently this kind of monitoring and preventative protection is at some level only in standardized intranet environments, not in internet connection level. However, some proactive means can be introduced to the MDM to increase security also against zero day vulnerability:

- Strong authentication keys
- Device applications / processes monitoring
- Application whitelisting (only allowed applications can run)
- Device physical configuration monitoring (e.g. USB ports)
- Connectivity and data transfer monitoring
- Internet connections security enforcement (e.g. only VPN connections allowed)
- Antivirus and other advanced threat protection applications
- Data encryption
- Network isolation / firewalls
- Connectivity monitoring
- Network anomaly detection
- Device recognition / monitoring
- Limit device operation / connectivity to certain physical location
- Network data transfer monitoring
- Network level antivirus scanning
- Forced RSA / strong key pair authentication
- Certificate based connections
- Forced data encryption
- Location / time based access rights to systems
- Proper Master Data Management

One extremely important aspect in cyber security is identification of the devices and users. The ID management system needs close integration to device management and also the technical means for authentication need to be at proper level, meaning use of device specific certificates. Provisioning methods and especially addressing the automation introduces challenges for proper scaling, security and avoiding "chicken and egg" syndromes in authentication and enrollment.

Also evolution of distributed ledger and block chain technologies for identity management will bring means for ensuring security in authentication and communication security in both cloud-to-device and device-to-device communication.

The Open Web Application Security Project's (OWASP) Internet of Things Top 10 Project aims to educate users on the main facets of IoT security and help vendors make common appliances and gadgets network- and Internet-accessible. The project walks through the top 10 security problems that are seen with IoT devices and discusses how to prevent them. These types of projects are just the beginning of the future security standards that must be developed to create a network of devices that benefits users in a secure environment. Many of these items can be taken into much better level with help of advanced device management system. The Top 10 Project items are:

1. Insecure Web interface
2. Insufficient authentication or authorization
3. Insecure network services
4. Lack of transport encryption
5. Privacy concerns
6. Insecure cloud interface
7. Insecure mobile interface
8. Insufficient security configuration
9. Insecure software or firmware
10. Poor physical security

## **Automation**

The IoT scenario “everything will be connected” means that even in home environments (not to mention offices, logistics, factories, etc.) there will be a tremendous need for device management but on the other hand scaling into that kind of multi-device / multi-platform / multi-form factor environments cannot happen if there is not intelligent learning, strong automation and other methods used for making the whole process of management automated.

In current management systems customization and adaptation are needed to each new device and platform type. That alone brings major roadblock in the speed and ability to adapt to changing environments.

Machine learning and artificial intelligence connected to device management tasks automation and administration alert systems will be mandatory in order to make proper management of large and heteronomous systems possible. This field is going to need a lot of research activities and standardization before it can be taken under proper development activities for device management systems.

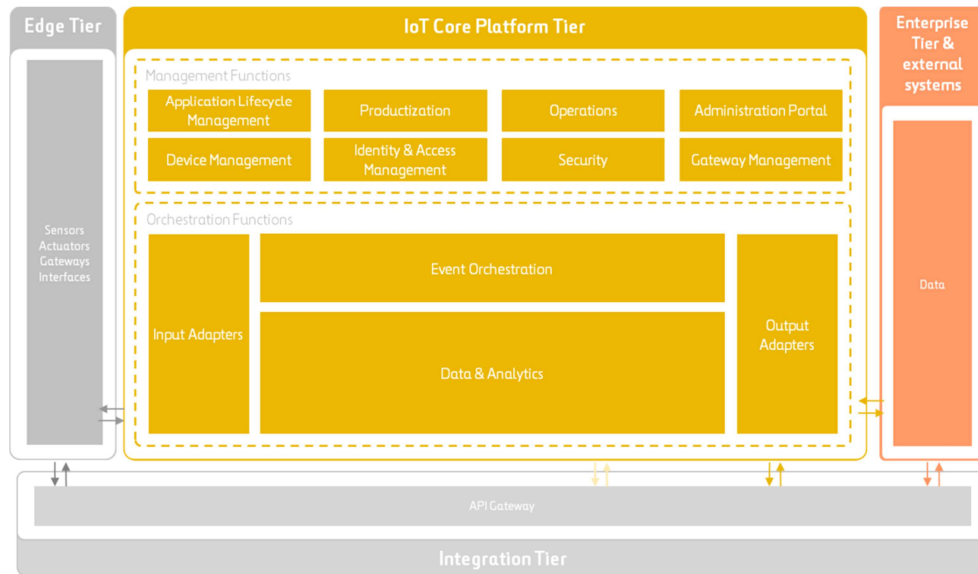
New functionalities in device management system is needed for monitoring and fetching and parsing log data and other information from target devices in order to make automated smart alerting and faster reaction or preventive actions to possible problems in field operations, especially in IoT but also in enhanced security mobile terminals.

## **Integration**

MDM/EMM systems are often necessary needed to integrate with other backend systems rather than run MDM as a standalone system. The backend system includes for example ID management, master data management, document systems, application management, and billing systems. In evolution of device management to IoT the need for integration will be

even greater as there are many more systems to integrate to, for example IoT platform, ERP, CRM, data analysis systems. Also in IoT the integration is tighter and in many cases the device management system needs to be controlled entirely from other systems.

IoT systems in general consist of multiple separate parts as depicted in Figure 21:



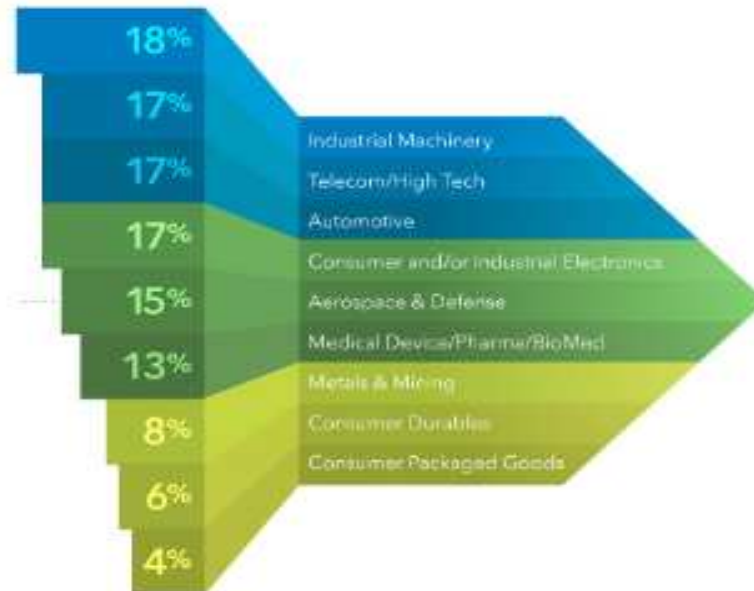
**Figure 21 IoT integration view**

There is a common understanding within the IoT industry that developing industry solutions need proper collaboration between the stake holders. Example of such collaboration is SuperIoT alliance (<http://www.superiot.fi/the-process>) started in Oulu. There are a lot of gaps still existing before IoT can be in mainstream [2].

### Commercial aspects

Market size for MDM systems will continue to grow slowly, mainly due to mobile device security needs of enterprises. IoT device management market has a huge growth potential in upcoming years in different market verticals. The uncertainty in that market is related to whether large IoT platform vendors like IBM, Amazon, Microsoft or ThingWorx will include device management functionalities as standard components in their platforms. From IoT platform perspective the device management is infrastructure and not as fancy as the data analysis and business intelligence which may lower the interest to develop such systems.

An estimation of the IoT market segments is presented in Figure 22.



**Figure 22 IoT market segments by SAS Software**

In general, any device management system provider will need much more flexibility in business models than in the past. Especially in IoT there is also need to productize customization process and service and consultation services. It can be also predicted that generic MDM system license price erosion will continue. Customized (hardened) mobile device management solutions will keep reasonable license prices but size and evolvement of that market is currently unknown. IoT device management license prices per device will be also very low due to very large device quantities. In smaller device systems the device management solution itself will have some price tag rather than per device licenses.

### **Conclusion**

Main items in future device management systems are related to scaling, customization, flexibility and integrations of the device management system. Main need and growth in device management business will be in IoT due to automation and cyber security needs in that field.

Cyber Security is one of main concerns and major roadblock for vast expansion in amount of connected devices. Security and IoT evolution will need standardization, alliances, good co-operation capabilities and close integration between different technology providers.

Evolution in enterprise device management will divide to large enterprise EMM needs and small and medium enterprise basic MDM needs. Organization needs are also divided into

basic security level needs and advanced security needs as for example in governmental organizations.

Another issue to be considered in future as IoT devices are adopted to organizations and proper responsibilities is how security and management is defined properly over whole life cycle of devices.

IoT and other connected device manufacturers will need proper and customizable device management tools for monitoring and updating their device fleets as well as keeping cyber security of the devices at acceptable level.

In technology perspective most development and evolution is needed in following areas: scaling, new communication channels/protocols, automation, intelligent/learning systems, authentication and cyber security in general.

Majority of these items are standard development activities for device management tools but especially interoperability and intelligent automation will need advanced research activities, standardization and new innovations like block chain utilization for identity management.

Market in MDM/EMM solutions is difficult due to commodization and heavy competition. Market for customizable niche MDM products might start growing and market for IoT device management is expected to grow heavily over few next years.

## **References:**

[1] [https://community.hpe.com/t5/Protect-Your-Assets/HP-Study-Reveals-70-Percent-of-Internet-of-Things-Devices/ba-p/6556284#.U\\_NUL4BdU00](https://community.hpe.com/t5/Protect-Your-Assets/HP-Study-Reveals-70-Percent-of-Internet-of-Things-Devices/ba-p/6556284#.U_NUL4BdU00)

[2] Julien Mineraud, Oleksiy Mazhelis, Xiang Su and Sasu Tarkoma:  
<https://arxiv.org/pdf/1502.01181v3.pdf>





**Where leaders and winners meet**

DIMECC Oy  
Korkeakoulunkatu 7  
33720 Tampere, Finland  
[www.dimecc.com](http://www.dimecc.com)