embeddedworld2015
Exhibition&Conference
... it´s a smarter world

IoT: The Internet of Threats and Static Program Analysis Defense

Dr Ralf Huuck
Red Lizard Software / NICTA
Sydney, Australia
ralf.huuck@redlizards.com

***Abstract*—By 2020 there will be an estimated 50 billion connected embedded devices in operation. This unprecedented large-scale machine-to-machine connectivity opens up a whole new set of challenges and security threats. Even common household devices such as smoke alarms, air conditioners or so called wearable health monitors pose a direct personal threat as well as a larger threat to infrastructure and economy. We highlight some of the threats coming from those IoT devices and explain how static code analysis will be one of the essential tools to assists to combat security issues right from the start of the software development lifecycle.**

Keywords—IoT; SDLC; DAST; security; static code analysis;

I. Introduction

While a few years ago the idea of Internet of Things was merely a concept confined to R&D labs and Universities this has changed dramatically by the advent of new products, added connectivity of existing products and the rise of big data with its corresponding information sourcing demands.

An estimated 50 billion devices will be machine-to-machine connected by 2020 and even today some blue chip companies such as Intel generating multiples of revenue from IoT compared to mobile and communications [7].

However, at the same time as the opportunities for IoT are rising the large-scale machine-to-machine connectivity opens up a whole new set of challenges and of security threats. Once even common household devices such as smoke alarms, air conditioners or so-called wearable health monitors are connected we will see direct personal threats as well as a larger threats to infrastructure and the economy. For instance, compromised smoke alarms can trigger chaos for emergency services, corrupted health monitors might lead to wrong medication regimes of individuals and hacked air conditioners might result in large scale power grid failures by simply switching all hacked devices on/off simultaneously.

In this work, we propose a specific software lifecycle solution to minimize the potential security threats and implications. In particular, we reason why *static program analysis* is one important tools for risk mitigation and security improvement. We explain, how these automated tools can achieve a high degree of confidence at relatively low cost, suiting the makers of low-cost embedded devices.

Based on our experience as a toolmaker and participant of various comparative studies as conducted by the US Department of Homeland Security and NIST, we highlight some of the key difference of those static analysis tools and their applicability to the Internet of Things. We give a number of concrete examples how these tools can help to create more secure software and we also show example software vulnerabilities from the field.

II. Increased Manufacturer Responsbibility

Traditional embedded devices used to be hardware components with small to medium size software drivers and applications to enable a limited interface to those components. Over the years that has changed drastically, with many embedded devices being essentially fully functional computers often running Android, Windows or Linux operating systems. As such the software components have grown almost exponentially in size making their correctness and security paramount.

In traditional settings with non-connected devices any perpetrator needs physical access to that device or its user interface to hack into the system. Moreover, for non-connected devices there is only limited danger of the spreading of an infection to other devices and systems.

Connectivity has changed the above scenario. If an embedded device is accessible over the network an attack can be remotely triggered often leaving the perpetrator anonymous and enabling him or her to attack from physically locations that makes prosecution difficult to impossible. Moreover, once hacked into a connected device there are possibilities for the attack to spread and take control over its connected components. This is what can be observed with common botnets.

IoT or machine-to-machine devices are affected beyond the base security risks mentioned above. Because there is no longer a human element in interacting with many systems, any intrusion detection is much harder, leaving the attacker potentially unobserved until a wider attack is triggered. Example security risks are connected home appliance sensors, which can potentially be used to siphon off user's private data undetected (such as in security cameras images or building occupancy information), or "captured" air conditioners, which can be used as a bargaining chip to extort utilities companies by threatening power surges. It is possible that an attacker gets into such a systems and remains undetected for long periods of time. Counter measure might be hard to deploy, because:

- IoT manufactures might not have the expertise to remove attackers from hijacked systems,
- IoT manufacturers might have gone out of business or stopped supporting platforms that are under threat, or
- IoT manufactures point to and blame other systems that supply faulty or compromised data to their own systems.

At the same time there is a key emerging burden to the manufacturer: IoT devices might not be updatable by the end user and as such shifting the sole responsibility back to the manufacturer. For instance, it is unlikely that consumers of smoke alarms or air conditioners can be asked to update their device firmware as they would for their laptops or mobile phones. They whole idea of IoT devices is the seamless integration into an infrastructure, which does not need to be managed by the consumer.

By shifting the burden to the device manufacturer additional questions arise: Are security breaches warranty issues? How to disable breached devices and what are the potential replacement costs? What is the level of software and network expertise required for manufacturers? All these questions will become relevant in the near future.

III. SDLC and Security

Security and in particular software security should be core to any systems development. IoT security spans across many components including the hardware, the network and most importantly the software stack that is commonly running on those devices.

In case of software, the software development lifecycle (SDLC) typically comprises software design, implementation, testing and release. There are various models ranging from traditional waterfall approaches to more common agile development approaches. In any case security is a challenge that needs to be considered in all of the aforementioned stages.

When it comes to security risks, it is well understood that half of all software related security vulnerabilities are related to software development bugs that could have been prevented in the implementation stage. However, with growing code bases and growing software complexity manual code reviews and testing are typically not enough to achieve a satisfactory level of confidence and security.

The need for automation as much as possible is obvious and various approaches to detect software security issues early in the SDLC have been developed. One of the increasingly popular solutions to scan source code for security vulnerabilities is automated static program analysis.

IV. Static Program Analysis For Security

Static program analysis subsumes a range of techniques from various fields of computer science to determine properties of a program at compile-time by analyzing the source code. This is unlike (dynamic) testing, where the program is executed to observe its behavior. Instead, a static analysis tool approximates the actual program behavior by building an internal model of the source code and applies different mathematically sound techniques as well as heuristics to identify specific coding errors that might lead to run-time errors such as crashes and memory leaks, or that might lead to security vulnerabilities. Examples of such security issues are SQL injections, tainted buffer allocations and format string vulnerabilities.

An example of such a vulnerability that can be detected by static code analysis is shown in Figure 1. The code intends to read in a user name for a database lookup. The input function gets() is called to obtain the name from the application user or some other communicating IoT components. This input is then used in a subsequent database query. However, at no stage the external input is sanitized. This means, the external component can write arbitrary text that will then be executed in the lookup query. In particular, the SQL queries themselves can be injected that will then be executed leaving the door wide open for potential exploits by querying for sensitive data or passwords.

```
void example(void * conn) {
  char *name;
  char *sql;
  name = gets(name);
  strcpy(sql, "SELECT age FROM people WHERE name = \"");
  strcat(sql, name);
  strcat(sql, "\"");
  sqlite3_exec(conn, sql);
                          }
```

Figure 1: SQL Injection Example

Certain static program analyzers such as Goanna by Red Lizard Software [2] are able to detect such unprotected use of external data and can automatically warn in those cases. An example output from a tool is shown in Figure 2.

```
Goanna[INJECTION-SQL]: User-controlled value `name' used
as part of an SQL statement in `sql'.
```

Figure 2: Example Goanna Warning

Another example is taken from the source code of a network tool called Wireshark. In the code fragment below there is an inconsistent use of the C pointer `nti`. In the second `case` instruction `nti` is checked against Null-ness, but in the first one it is not. This means, if `nti` can indeed be Null, it would trigger a crash leading to a potential exploit.

```
case NT_TRANS_IOCTL: (…)
  dissect_smb2_ioctl_data(ioctl_tvb,pinfo,tree,top_tree,
  nti->ioctl_function, TRUE);
  (…)
case NT_TRANS_SSD:
  if(nti){switch(nti->fid_type){ (…)
```

Figure 3: Potential Null Pointer Exploit

The Goanna program analyzer is able to detect these inconsistencies automatically, which otherwise would be very laborious tasks for a human. Goanna reports:

```
Goanna[NULL-Possible]: Parameter `nti' is not checked
against NULL before it is dereferenced on some paths, but
on other paths it is.
```

Figure 4: Goanna Null Warning

It is unlikely that traditional testing alone can unearth some of the examples above, since they are only triggered along certain execution paths when using very particular data.

The advantages of code analysis tools such as Goanna is that they are fully automatic and can be run at any time the actual source code fragment in question compiles. This reduces not only the burden on the security analysis teams and testers, but also significantly cuts down on software lifecycle costs, as it moves security forward to be a pro-active activity of the software developer. As such, the economic argument for tool usage is rather straightforward.

V. TOOLS AND STANDARDS

There are is range of software security code analyzers with different levels of strength. While Goanna is a security analyzer based on formal verification techniques for C/C++ [3] there are other tools using different approaches or targeting different programming. First steps are made in US Government projects such as SAMATE and the DHS funded software assurance market place (SWAMP) to harmonize tool output and to classify different tools and their capabilities [4].

Moreover, there are a range of security measures and standards out there that play a critical role in assessing a company’s code security. The base classification for many software bugs is collected in the Common Weakness Enumeration (CWE) set as maintained by the MITRE organization [5]. This lead to a number of security specific classifications including CERT by the Software Engineering Institute of Carnegie-Mellon, SANS by the same named organization, and OWASP by a community focusing on web security. These classifications also form the base for the upcoming ISO/IEC 17961 standard [6].

VI. SUMMARY

IoT devices enable tremendous opportunities to automate common task by increased machine-to-machine interaction. However, there are novel security risks that are affecting the business of IoT manufacturers in an unprecedented manner. The additional responsibility to deal with unknown modes of device interaction makes security a hard challenge. Automation will be a key component in assisting to prevent these threats, especially when considering larger and more complex growing software. Based on our experience and work in various international efforts we see static program analysis as one of the key components to complement the security tool landscape. As shown in the article, an analyzer such as Goanna is able to significantly assist IoT manufacturers to create more secure software without compromising development processes and productivity.

ACKNOWLEDGMENT

This work has been supported by NICTA. NICTA is funded by the Australian Government (Department of Broadband, Communications and the Digital Economy) and the Australian Research Council through the ICT Centre of Excellence program.

REFERENCES

[1] John Viega & Gary McGraw. Building Secure Software. Addison-Wesley - ISBN 0-201-72152-X.

[2] Ansgar Fehnker, Joerg Brauer, Ralf Huuck, Sean Seefried. Goanna: Syntactic Software Model Checking. 6th International Symposium on Automated Technology for Verification and Analysis (ATVA), October 20-23, 2008 Seoul, Korea.

[3] Ralf Huuck, Ansgar Fehnker, and Rodiger Wolf. Model Checking Dataflow for Malicious Input. Proceedings of the 6th Workshop on Embedded Systems Security Taipei, Taiwan, Oct 2011. ACM, Article 4, 10 pages, ISBN: 978-1-4503-0819-9.

[4] Software Assurance Marketplace. http://continuousassurance.org/

[5] Common Weakness Enumeration. http://cwe.mitre.org/

[6] ISO/IEC TS 17961:2013 Information technology -- Programming languages, their environments and system software interfaces -- C secure coding rules

[7] URL:http://www.theverge.com/2015/1/16/7555647/the-internet-of-things-is-already-a-2-billion-business-for-intel