DTM components: shadow keys to the ICS kingdom

16.10.2014

This document is a preliminary revision.
For the latest version, please see
http://github.com/Darkkey/DTMResearch

Authors:

Alexander Bolshev
(@dark_k3y, abolshev@dsec.ru)

&&

Gleb Cherbov
(@cherboff, g.cherbov@dsec.ru)

Digital Security Research Group

Table of Contents

0. Intro: ICS 101

ICS stands for Industrial Control System. This is a general term which unites software and hardware complexes used in industry automation, including distributed control systems (DCS), supervisory control and data acquisition systems (SCADA), and other systems, such as PLC, HMI, MES, etc. Today, ICS infrastructures are commonly used in every factory and even in your house, too! Simplistically, ICS collects data from remote stations, processes them and uses automated algorithms or operator-driven supervisory to create commands to be sent to remote devices (also called field devices). Field devices are doing the dirty work, such as condition monitoring, starting and stopping engines, collecting data from sensors, etc.

Modern ICS infrastructures have complex architectures that consist of commonly known parts such as servers, PCs, network switches, software technologies (.Net, DCOM, XML, etc.) and a bit scary things such as PLCs, transmitters, actuators, analog control signals, etc. This unity allows us to attack such systems in various ways. In this whitepaper, we will talk about how we researched some of ICS infrastructure chain links. About certain weaknesses in the middle-level parts of the ICS (level of SCADA and HMI components) which could be exploited from the low level (level of field devices and field interconnections) to trigger weakness in the high parts of the ICS (MES, PAS, and other software). We will walk through the “back door” of the ICS to receive our keys to the kingdom.

As we have said earlier, modern ICSs are complex systems which integrate many technologies and architectures. Let’s review a sketch of the modern ICS infrastructure (see fig. 1). We can see three basic levels of the ICS on this figure:

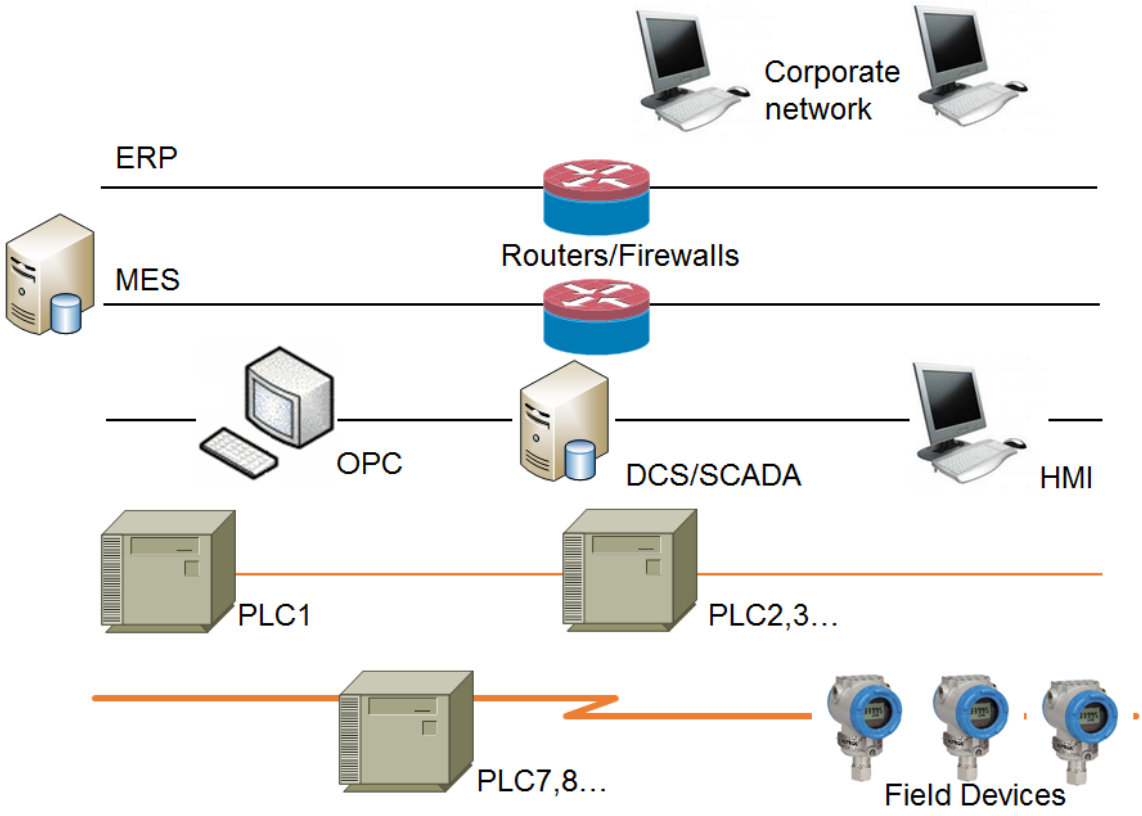

Figure 1: Modern ICS infrastructure sketch.

1. **Lower level**, where field devices exist. As is was mentioned earlier, these devices are suitable for the dirty work – for example, they can monitor temperature and pressure in a reactor, control operations such as opening and closing valves, turning on and off pumps, and other things. Multiple devices can be used on this level. It may be a low-level Programmable Logic Controller (**PLC**, according to Wikipedia[1]: computer-based, solid-state device that controls industrial processes). Also, transmitters and actuators, controlled by a Remote Terminal Unit (**RTU**, microprocessor-controlled electronic device that interfaces objects in the physical world to a distributed control system or a SCADA (supervisory control and data acquisition) system by transmitting telemetry data to a master system, and by using messages from the master supervisory system to control connected objects[2]), can be used on this level. This level is the kingdom of low-level industrial protocols, such as Modbus or Modbus TCP, HART, Wireless HART, Profibus DP or PA, Foundation Fieldbus H1, and others. Low-level ICS engineers, electricians, technicians, and other staff are working with ICS on this level.
2. **Medium level**, where we have high-level PLCs, Distributed Control System (**DCS**, typically employed to control a manufacturing system, process, or any kind of dynamic systems, in which the controller elements are not central in location (like the brain) but are distributed throughout the system with each component sub-system controlled by one or more controllers[3]), Supervisory Control and Data Acquisition systems (**SCADA**, systems operating with coded signals over communication channels so as to provide the control of remote equipment (using typically one communication channel per remote station)[4]), and Human-Machine Interface (**HMI**) controlling stations and various servers such as Open Platform Communications (**OPC**, earlier OLE for Process Control). All intellectual work takes place on this level. Here, based on the data from lower levels, operators or automated systems make decisions on what to do and send commands to field-level devices. The entire industrial automation process occurs here. Operators, process engineers, ICS engineers, PLC and software programmers are working with systems on this level.
3. **Upper level** refers to the integration of business with industrial processes. This layer provides bindings to the corporate networks and Enterprise Resource Planning (**ERP**) systems. Various Plant Asset Management (**PAS**) and Manufacture Execution Systems (**MES**, which provide the right information at the right time and show the manufacturing decision maker *how the current conditions on the plant floor can be optimized to improve production output*[5].) work on this level. Management and top-level engineering staff work with ICS on this level.

Now, you may want to ask the question: "How are such different systems interconnected with each other?". To list the most commonly used technologies in ICS:

- Windows
- Linux
- Ethernet
- HTTP
- XML

[1] http://en.wikipedia.org/wiki/Programmable_logic_controller
[2] Gordon R. Clarke, Deon Reynders, Edwin Wright, Practical modern SCADA protocols: DNP3, 60870.5 and related systems Newnes, 2004, ISBN 0-7506-5799-5, pages 19-21
[3] http://en.wikipedia.org/wiki/Distributed_Control_System
[4] http://en.wikipedia.org/wiki/SCADA
[5] McClellan, Michael (1997). Applying Manufacturing Execution Systems. Boca Raton, Fl: St. Lucie/APICS

- DCOM
- .NET
- SOAP
- SQL

Looks familiar, isn't it? Add the following ICS-specifics terms to the above list:

- Industrial protocols (DNP3, Modbus/Modbus TCP, Profibus DP/PA, Foundation Fieldbus, HART, Profinet, etc.)
- OPC & friends
- FDT/DTM
- Many other things…

The answer to your question is "**deep integration**". At some point in the past, ICS vendors understood that they should work together. Various standards and specifications were created to help devices, software, and technologies from different vendors work with each other. We have tons of IEC specifications, we have OPC foundation, FDT/DTM consortium. Also, many vendors produce custom technologies and methods to integrate with others vendors' systems. We all know that **deep integration** is not very efficient without **deep trust**. And this will be our attack point.

Today is not the 199x/200x, when every ICS infrastructure was highly insecure. Unpatched systems, web interfaces of PLC and SCADAs being open to the Internet, total mesh of ICS network and corporate network – most of this has gone to history. For the old infrastructures that cannot be upgraded immediately, ICS engineers have industrial firewalls, special SIEM systems, "diodes" systems, special IDS/IPS configurations. However, a common mistake when you are establishing the security of ICS infrastructure is to forget about the lower level.

Even in the modern ICS architecture, the lower level is driven by old and dirty industrial protocols. Such protocols have scarce security mechanisms or none at all. We are talking about such low-level protocols as Modbus over RS-485, HART, Profibus DP and PA, Foundation Fieldbus H1, and others. These protocols are not based on the Ethernet or modern wireless stacks; they are based on technologies invented back in the 197x/198x . This can be useful in some specific industrial facility areas where electromagnetic or other noises can corrupt or disrupt modern L1/L2 network technologies. However, this is a rare case, and in most parts of the industry such old protocols are just legacy. So, if you find the way (using some analog or digital electronics magic) to connect to a live low-level industrial protocol and overcome certain technology "features", you can sniff and inject your own packets in this network. Sometimes it's easy, sometimes it's hard, but it's always **possible**. However, other devices and applications which reside on this line don't expect it. They can be vulnerable to specially crafted packets and instructions – because, due to deep integration and deep trust, their developers (in some cases) didn't think about receiving incorrect packets (yes, they developed mechanisms to evade *corrupted* packets, but they didn't expect packets with *correct CRC and incorrect content*. Moreover, some information collected at the field device layer is passed to the higher levels: not even to the middle layer of SCADA and OPC, but to the layer of MES and even ERP! And this is a very interesting "feature" that can be used to attack not only the lower layers of network and/or industrial processes, but also corporate networks. And the first step for an attacker on their way from the entry point (field devices) is to breach the middle level applications. Let's see what it looks like!

1. Intro to FDT/DTM

Imagine a modern industrial facility – power plant, for example – and think about how many different field devices exist in its infrastructure. There could be several thousands of such devices just in one division of the facility. These devices are from different vendors, work over different low-level protocols, and are organized into complex hierarchical networks. Now ask yourself: how to support all these devices? Even walking to each of them with a handheld communicator could be painful. The other task appears when you are creating a DCS system with such devices – different vendors could use the same low-level protocol but custom commands, extensions, etc. When you develop your SCADA, you will need to learn all such details. Looks awful, right? The solution to these problems exists and is called FDT/DTM specification.

FDT Joint Interest Group was founded by such companies as ABB, Endress + Hauser, Invensys, Siemens, and Metso Automation, in the early 2003. These companies now form the Steering Committee of the FDT JIG. FDT Joint Interest Group was a non-profit international business cooperation in the field of industrial automation. With a growing number of members, the management of FDT Joint Interest Group became increasingly difficult, and they decided to regroup as a legally independent organization. For this reason, **FDT Group** was founded as an association in September 2005.

FDT (Field Device Technology, formerly Field Device Tool) is a specification for a software interface. This software interface describes data exchange between field devices and drivers for a common framework application. International standards IEC 62453 and ISA103 describe FDT. The FDT/DTM consists of two parts: Field Device Tool / Device Type Manager. *The FDT concept defines the interfaces between device-specific software components provided by the device supplier and the engineering tool of the control system manufacturer. The device-specific software component is called DTM (Device Type Manager)*[6], the maintainer of FDT/DTM specification. In short:

- FDT standardizes the communication and configuration interface between all field devices and host systems
- DTM provides a unified structure for accessing device parameters, configuring and operating the devices, and diagnosing problems

Usually, a common framework application (FDT Frame application) acts as a container to all DTM components. Field devices can be configured, monitored, maintained from one software system, without paying attention to the specific model, type or industrial protocol of the field device. An FDT Frame application allows the engineer to load and create hierarchies of DTM device drivers and UIs. A Frame application can be PAS/AMS, DCS, or a simple device configuration tool. Plant Asset Management Systems (PAS or AMS) allow engineers to configure and monitor all field devices, and managers to review the full state of the field device infrastructure. PAS solves complex problems of field device management and integrates with high-level systems such as MES and ERP. One of the most popular FDT Frame applications is Endress & Hauser FieldCare. It provides multiple features, including integration with MES and ERP, binding to databases, data exchange with external applications, and the Condition Monitoring component, which allows monitoring a field device infrastructure using a web browser. In figure 2, you can see the FieldCare main window interface.

[6] FDT Group, http://fdtgroup.org

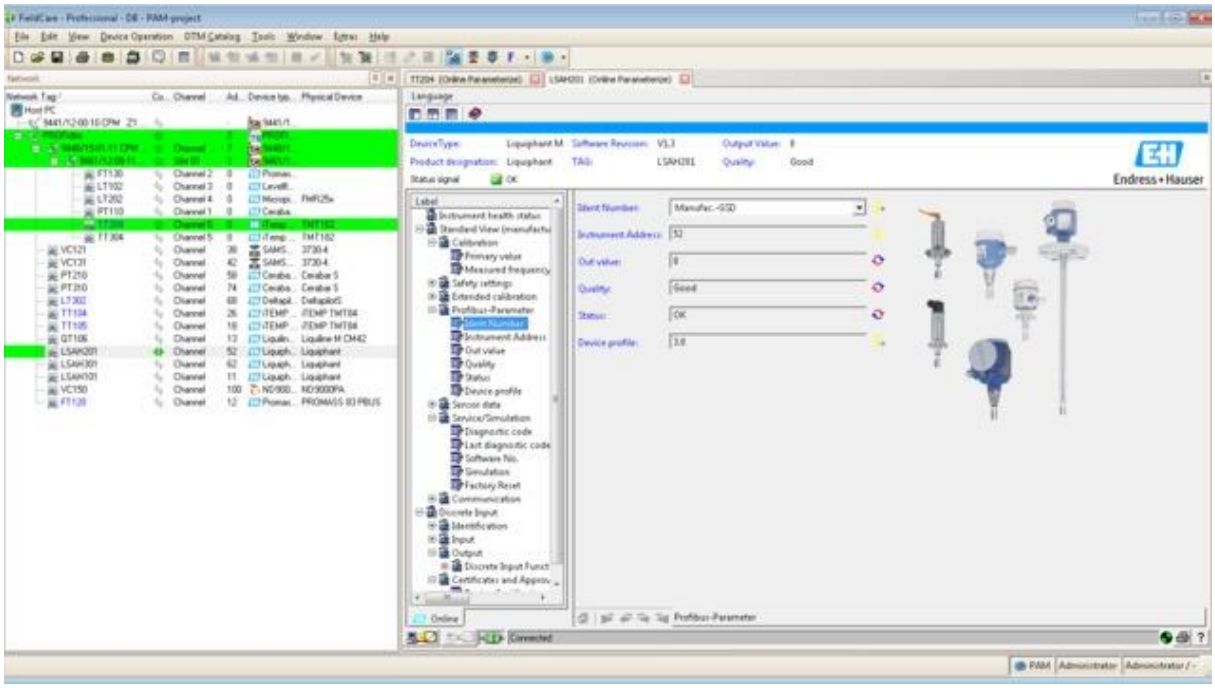

Figure 2: E&H FieldCare main window with several DTM components hierarchy.

DTMs can be created in different ways: as an FDT interface to existing standalone tool, as a generic interface to the Device Description Language (DDL) files, and as a device-specific DTM, “from scratch” (see figure 3):

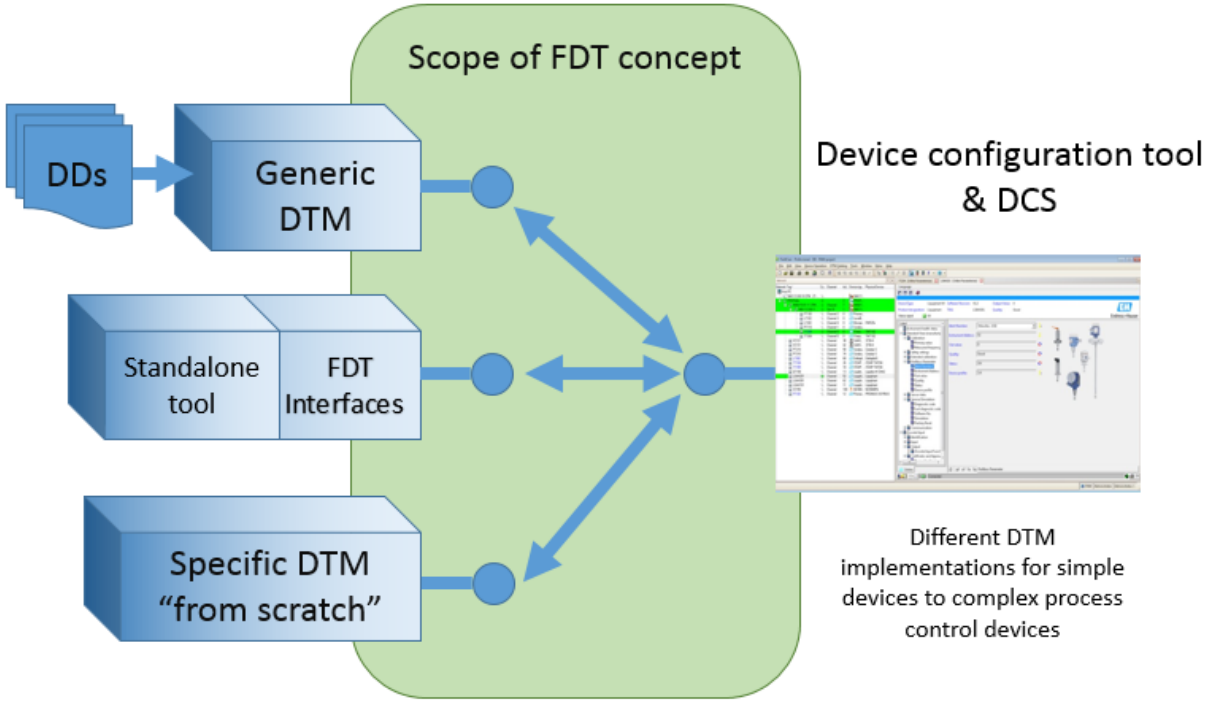

Figure 3: Different implementations of DTMs.

There are different versions of FDT/DTM, and we will stick to 1.2.1, which is currently the most popular one. FDT 1.2.1 technology is based on the interoperation of COM components. FDT frame application and various DTMs are exchanging XML messages with each other.

Basically, there are two types of DTMs:

- CommDTMs, which allow operating a specific industrial protocol
- DeviceDTMs, which provide the functionality to work with specific field devices

Combinations of DTMs can create complex hierarchical networks, where one DTM acts as the CommDTM for low-level DTMs and as the DeviceDTM for high-level DTMs. The basic hierarchy is shown in figure 4.

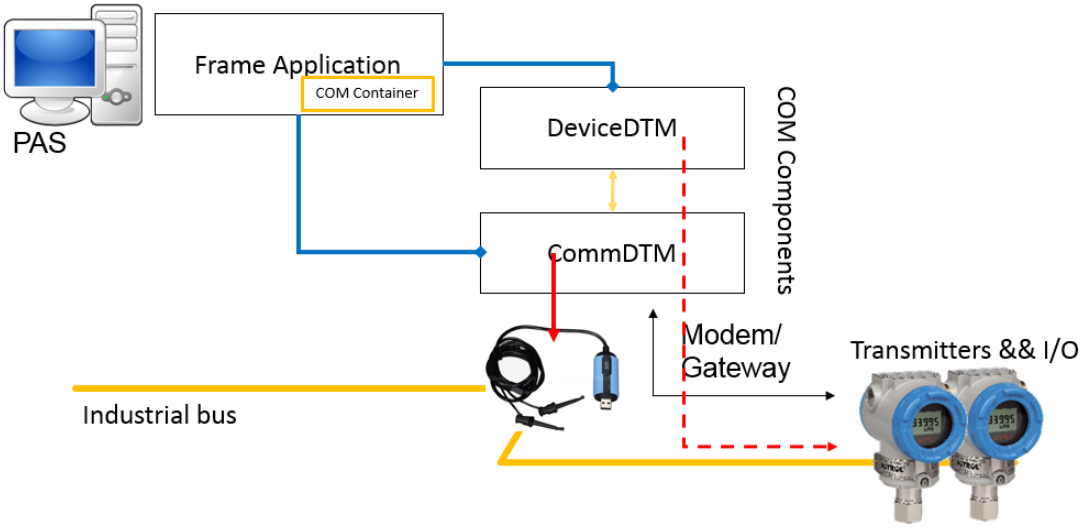

Figure 4: Basic DTM components hierarchy.

Here, FDT Frame contains two components. The CommDTM is responsible for communication with industrial bus through modem. The DeviceDTM is for communication with transmitters through the CommDTM. The hierarchies could be much more complex, as shown in figure 5:

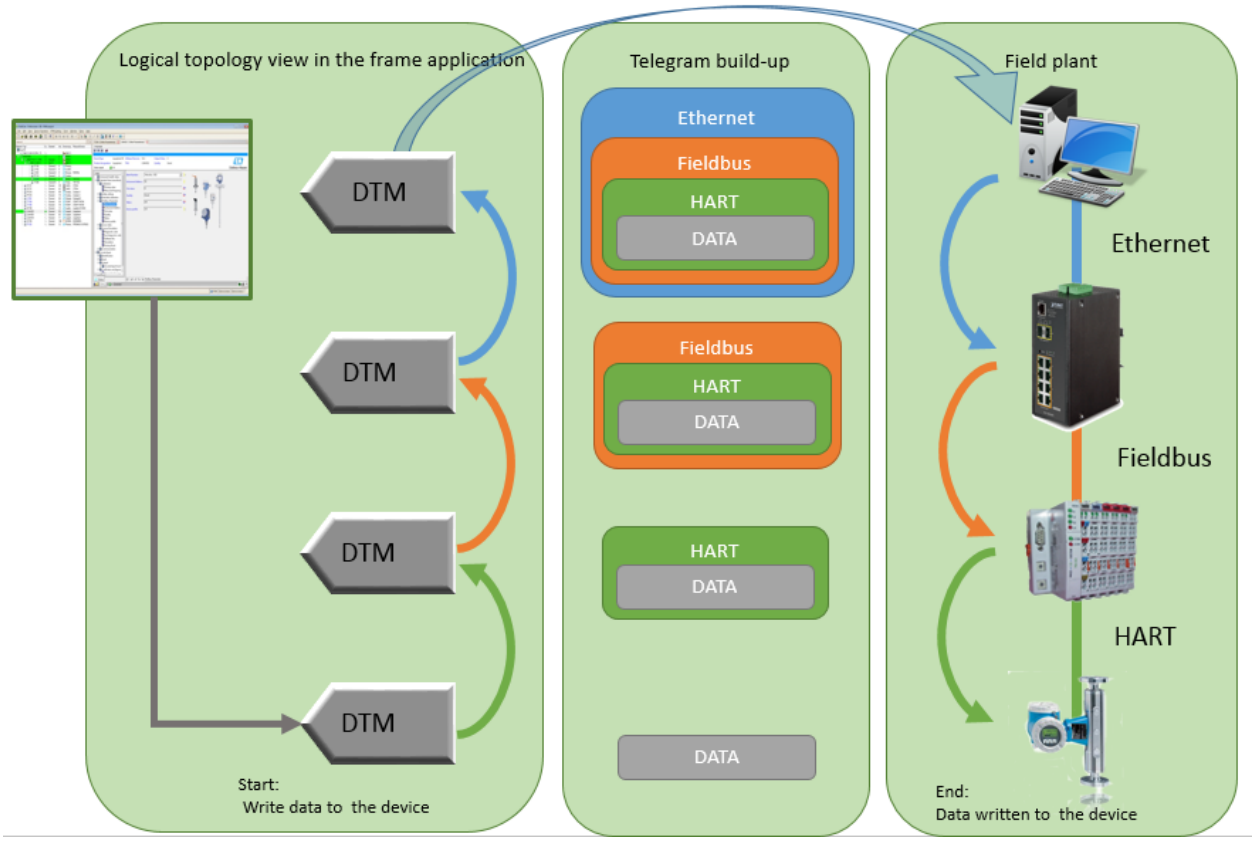

Figure 5: DTM communication in complex networks.

Simplified FDT architecture internals are shown in figure 6:

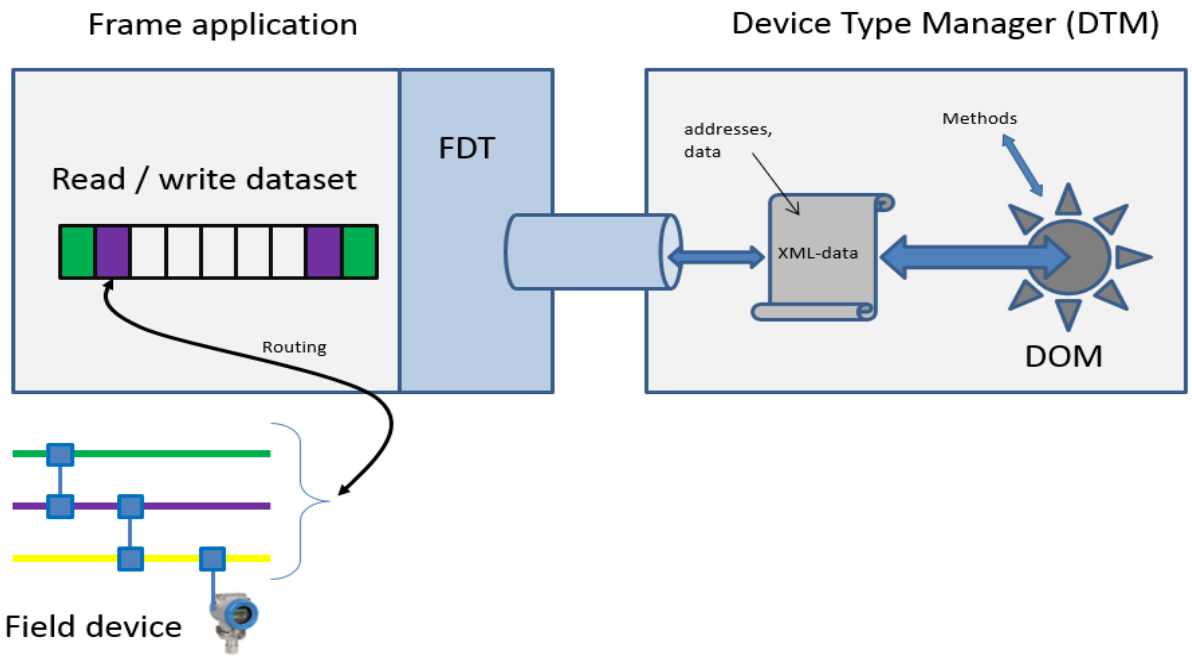

Figure 6: FDT/DTM architecture.[7]

Let's review the **simplified** process of interoperation inside FDT using simple sample. Imagine that we have a hierarchy like in fig. 6. In our facility, we have several field devices based on HART protocol that are connected to the current loop line. A server with FDT Frame is also connected to the current loop using a HART USB modem. What would happen if the engineer wanted to read data from a specific field device and clicked "Read data from device" in the Frame application menu? DeviceDTM will get the DownloadRequest() call from FDT frame. Now, using FDT API, DeviceDTM will ask FDT Frame for a pointer to the CommDTM and start setting up communication with it. After a few calls and callbacks, DeviceDTM sends a connect request to the CommDTM:

```
<?xml version="1.0"?>

<FDT xmlns="x-schema:FDTHARTCommunicationSchema.xml" xmlns:fdt="x-
schema:FDTDataTypesSchema.xml">

      <ConnectRequest fdt:tag="AAAAAAAA">

            <ShortAddress shortAddress="0"/>

      </ConnectRequest>

</FDT>
```

CommDTM will create a HART command: a packet to the device with a specific Polling ID. The packet will be sent to the current loop device using a HART modem. When the device answers and CommDTM gets the HART Unique ID, it will use the FdtCommunicationEvents.OnConnectResponse callback to notify DeviceDTM that all is ready for data transmission:

[7] Figure is the property of the FDT group and can be found in their official specification, see http://www.fdtgroup.org/technical-documents.

```
<FDT xmlns="x-schema:FDTHARTCommunicationSchema.xml" xmlns:fdt="x-
schema:FDTDataTypesSchema.xml">

<ConnectResponse fdt:tag="AAAAAAAA" preambleCount="0"
primaryMaster="1" communicationReference="1B36D3D1-C4DD-4BAF-9A31-
016F704E21F2">

<ShortAddress shortAddress="0"/>

</ConnectResponse>

</FDT>
```

Now, DeviceDTM will create a TransactionRequest with the exact command (and possibly data) that should be sent to the device:

```
<?xml version="1.0"?>

<FDT xmlns="x-schema:FDTHARTCommunicationSchema.xml" xmlns:fdt="x-
schema:FDTDataTypesSchema.xml">

     <DataExchangeRequest commandNumber="0"
communicationReference="1B36D3D1-C4DD-4BAF-9A31-016F704E21F2">

     </DataExchangeRequest>

</FDT>
```

CommDTM will parse the request, create a HART packet with the command and encapsulated data (if any), and send it to the current loop line. When the answer from the device is received, it will be parsed, the HART header and CRC byte will be removed, and the packet command data will be passed to the DeviceDTM using the IFdtCommunicationEvents.OnTransactionResponse callback:

```
<?xml version="1.0"?>

<FDT xmlns="x-schema:FDTHARTCommunicationSchema.xml" xmlns:fdt="x-
schema:FDTDataTypesSchema.xml">

 <DataExchangeResponse   commandNumber="0"
communicationReference="1B36D3D1-C4DD-4BAF-9A31-016F704E21F2">

<fdt:CommunicationData
byteArray="FE006209070201010110F01C070262000000C7000000"/>

<Status deviceStatus="0">   <CommandResponse value="0"/>  </Status>
</DataExchangeResponse>

</FDT>
```

Note two important things here:

1. All data is going to use XML. The device tag (fdt:tag) is the unique ID of the field device (and the DeviceDTM instance) in the FDT Frame hierarchy; the tag is assigned using **data from the field device** (e.g. for HART devices, it could be HART tag or HART long tag)
2. DeviceDTM doesn't directly talk to the device. Instead, it receives an **XML binary stream** from the device, and CommDTM only strips the packet header and tail

Also, there are more weak points of FDT/DTM environment: many DTM components were created for early versions of the specification or just as interfaces to existing libraries or software. All of this causes a really dangerous and vulnerable mix of such technologies as:

- OLE32
- ActiveX
- Visual Basic 6.0
- .Net
- COM
- XML

As you can see, different DTMs could use different thread models, different .Net versions, different XML parsers, different C++ runtimes, and all this crap has to work together. Thus, too many exceptions are ignored or passed without a handle. This could create a nice attack surface.

The last thing to talk about in this chapter: the place of FDT and DTMs in the ICS architecture. A nice picture (fig. 7) is available at the FDT group site:

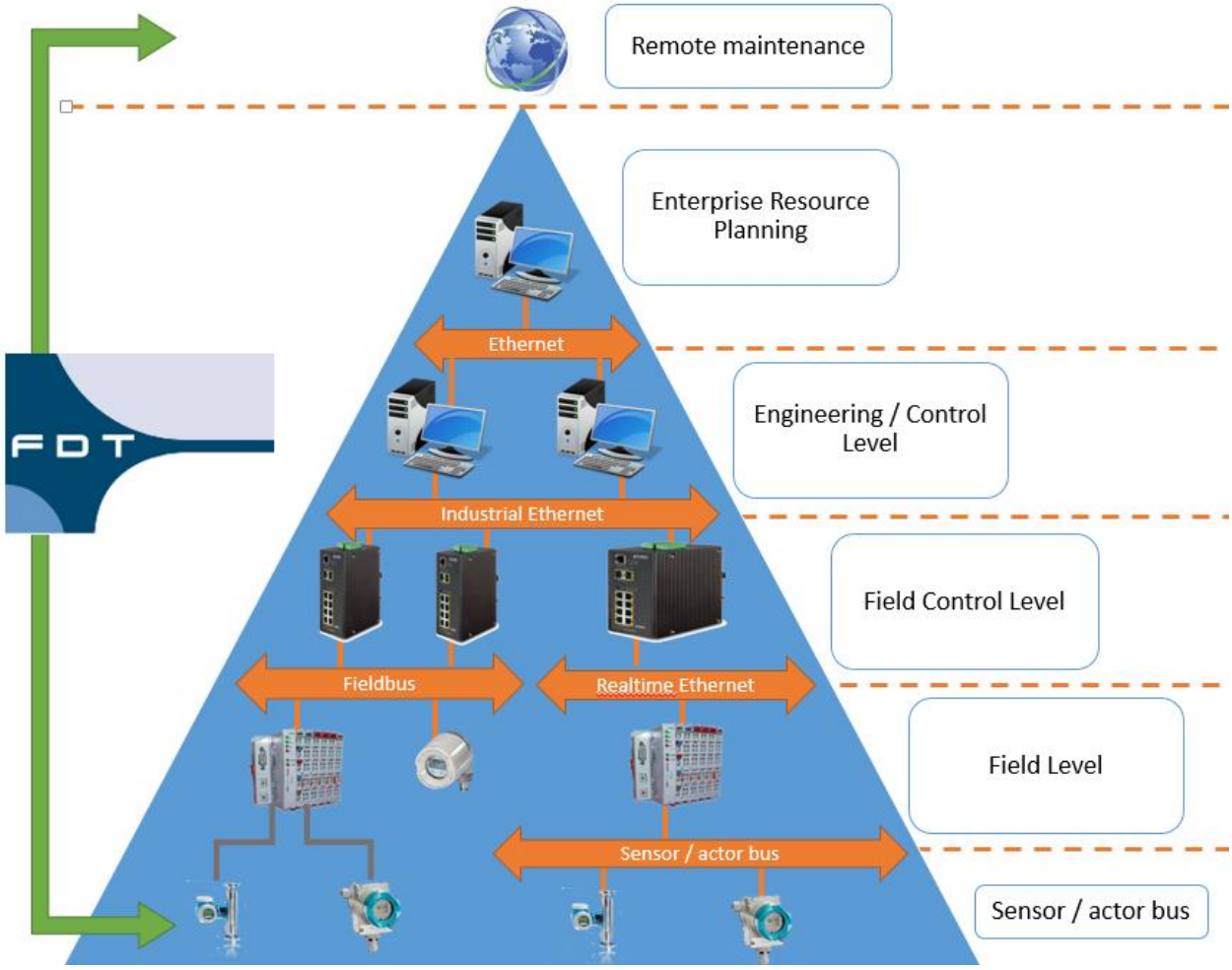

Figure 7: FDT and ICS levels.[8]

As you can see, FDT covers all levels of industrial facility. You could find DTM components in:

- PAS/AMS systems
- HMI systems
- DCS systems
- OPC servers

Data from DTM components can go to the upper levels, e.g. MES and even ERP systems.

[8] Picture from http://www.automationworld.com/fdt-group-wants-your-input-yes-yours

2. Research scope

If you look up the official FDT group site (http://www.fdtgroup.org), you will find an FDT components catalogue[9]. There are DTM components for thousands of field devices there. When we started our research of FDT, we understood that we needed to establish specific tasks and scope, because the whole scope is really big and can't be resolved by a small team of researchers. So, we decided that we wanted to answer three questions in our research:

1. Why is FDT/DTM architecture weak?
2. What kind of vulnerabilities in DTM components could cause a compromise of ICS infrastructure?
3. What about FDT 2.0 security?

The answer to the first question was partially given in the last paragraphs of the previous chapter. To give a complete answer and an answer to the second question, we need to look at some DTMs implementations. So, we need to take a sample of all DTMs and find out how much of them have weaknesses and/or vulnerabilities. This sample should not be very big, but it shouldn't be too small either, otherwise it wouldn't be representative. Thus, we've decided to research about 100 DTMs. The last question about the sample was "which low-level protocol should we choose when researching DTMs?" The answer was obvious for us: HART.

Finally, we had chosen 114 DTMs from 24 vendors for 752 HART devices.

Why did we choose only the DTMs for HART devices? For several reasons:

- We are familiar with this protocol
- We have hardware tools to work with and attack HART devices
- HART is used in critical industries, such as power plants, chemical factories, oil & gas, etc.

Let's talk a bit about HART. HART (Highway Addressable Remote Transducer Protocol) is one of the first implementations of field bus protocols. The biggest advantage of HART is that it can communicate over the standard 4-20 mA current loop. In the past, 4-20 mA current loop was used for receiving and sending information to a distant transmitter or actuator, using analog (current) signal level. When HART appeared (mid-1980s), it provided the capabilities for configuring distant devices over current loop and getting/sending additional information to them, rather than reading (or writing) only one variable.

HART is a master-slave protocol. Field devices (transmitters and actuators) appear on the network as slaves, whereas computers with HART modems, HART gateways, and PLCs with HART modules are masters. Generally, a slave cannot send any packets to the networks other than replies to master. According to the HART specification, there can only be one primary master (PLC, gateway, or server) and only one secondary master (PC or HART communicator).

HART uses two different addressing schemas for field devices: logic (Polling) and hardware address. Every field device has a unique hardware address (Unique address). It consists of Manufacture ID, Device ID, and Unique Device ID. Think of it as a MAC address. Also, in the digital-only mode, every field device has a Polling ID address, an integer in the 1-63 range.

[9] http://www.fdtgroup.org/product-catalog/certified-dtms

HART can work in two different modes: digital/analog and digital only. In the first mode, so-called point-to-point, digital signals are sent over a 4-20 mA analog current loop. The master host can receive a variable value traditionally, using analog current value, or based on digital HART packets. In this mode, the device's Polling (logical) address is always fixed to 0. In the second mode (digital, or multidrop), multiple devices exist on the line, and every device has its own Polling ID. When a master starts working with a device, it will call it by the Polling ID address to determine the field device unique address. Then, the master will talk to the device personally, using its Unique ID.

In HART FSK, all data is sent using a simple Frequency Shift Keying modulation, where '1' is encoded by one 1200 Hz harmonic and '0' is encoded by two 2200 Hz harmonics. For more information, see fig. 8.

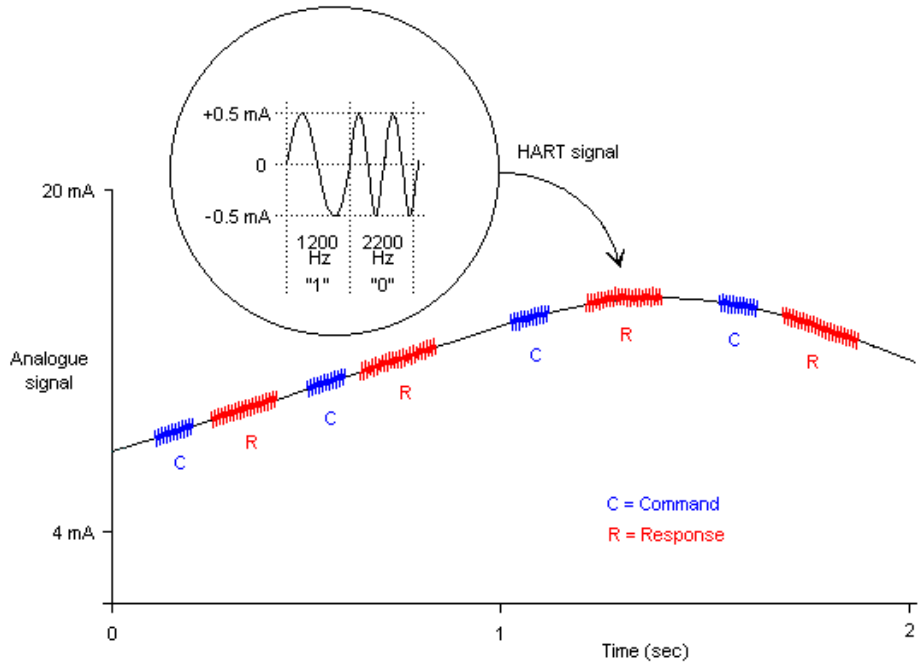

Figure 8: HART FSK[10]

Currently, the HART protocol is supported by HART Communication Foundation, see their website http://www.hartcomm.org/ for more detail on this protocol.

In table 1, you can see HART protocol communication levels according to the OSI layers:

OSI Layer		HART
7	Application	Hart commands
2	Datalink	Binary, Master/Slave protocol with CRC
1	Physical	FSK via copper wiring, wireless, RS-485, HART-IP

Table 1: HART communication layers

The physical layer is enabled by HART FSK. HART packets structure on the datalink layer is shown in table 2:

[10] Property of HART Communication Foundation, see http://en.hartcomm.org/hcp/tech/aboutprotocol/aboutprotocol_how.html.

Delimeter	Address	[Expand]	Command	Byte Count	[Data]	Check byte

Table 2: Datalink structure of a HART packet

On the application layer, there are tons of HART commands that can be divided into three categories:

- Universal (operations with ID, getting variables, setting variable limit and types, tag operations, etc.)
- Common practice (engineering and process-specific commands)
- Device Families (device family and vendor-specific commands)

Additional addressing scheme is introduced on the application layer of the HART protocol – tag and long tag. Tag is a packed ASCII string that provides an ID which is easy to remember for an engineer. The maximum tag length cannot exceed 8 symbols (6 packed ASCII bytes). When the number of field devices on the facility is increased, engineers need more convenient IDs, so the long tag command was added. The long tag allows storing a 32-byte ASCII string on devices and returning it to masters. As we've said earlier, this tag or long tag is commonly used by PAS/AMS, OPC servers, MES, and other software to identify field devices in the industrial facility structure.

Yet another reason for selecting HART DTMs is HART physical security issues and possible security issues in the encapsulation mechanism (that can refer either to HART or to FDT or to the whole ICS building principles). Let's review attack models for HART/FDT infrastructures. Imagine we have a DTM with a vulnerability. This vulnerability could be triggered by some evil data inside a HART packet. How can this data be landed there?

There are several ways for this to happen. The first is injection on the lowest level, e.g. an injection directly into the current loop line. Using some methods, you could achieve successful forging of the field device answers in the current loop line[11]. An attacker MitMs the real transmitter, forges it, and answers DTM requests with specially crafted packets, like in figure 9:

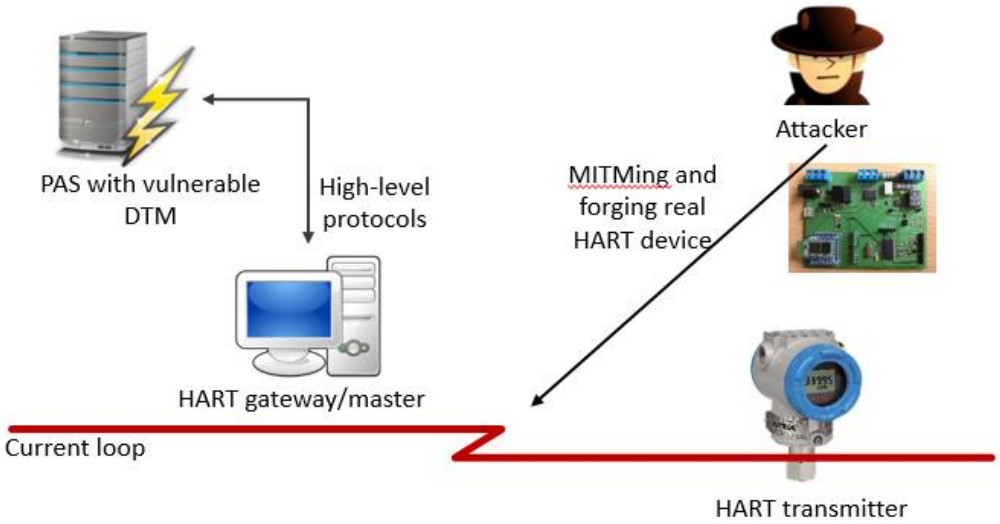

Figure 9: Attacking vulnerable DTM through current-loop line.

[11] See the HART (in)security: how one transmitter can compromise whole plant and HART as an attack vector: from current loop to application layer presentations.

This could be easier than it seems: HART current loop line length can reach up to 3 km, and sometimes you can find HART transmitters outside of the plant building – for example, pressure and corrosion transmitters on pipelines.

As we've mentioned earlier, data can go after the HART gateway into networks with other protocols. For example, it could be regular Ethernet, Wi-Fi or some other wireless channel. In that case, data can be repacked into another format, like HART-IP, HART over TCP/IP. This could increase the attack surface, e.g. in HART-IP, packet length is coded by two bytes (maximum value = 65535) instead of one in HART (maximum value = 255). The DTM component may be surprised with such amount of data. The attacker in this situation is using standard tools and principles for MitMing hosts in an Ethernet network and then forging packets from one gateway to the next gateway, as shown in figure 10:

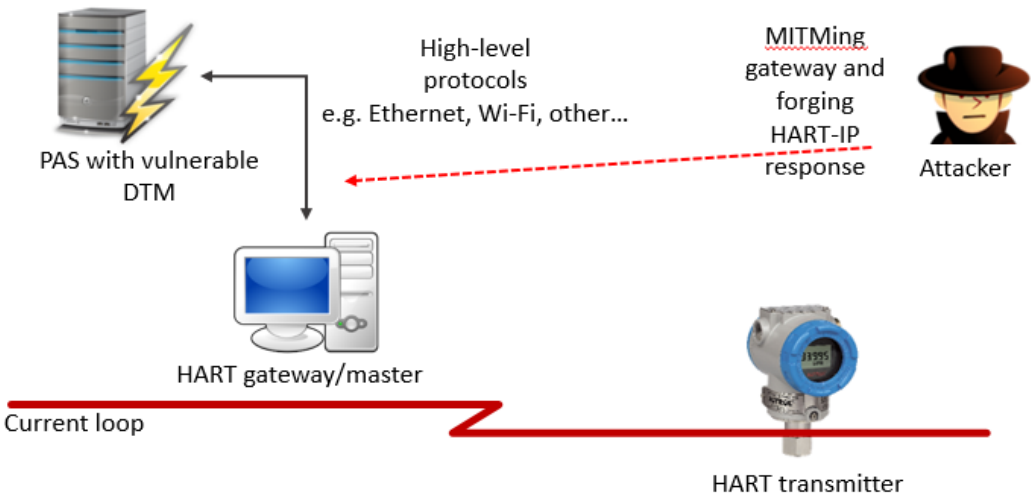

Figure 10: Attacking vulnerable DTM through upper levels (HART-IP).

The third case is an attack on one of the industrial protocols in the packet transmission chain. For example, it can be an attack against Modbus or Profibus-DP (see figure 11):

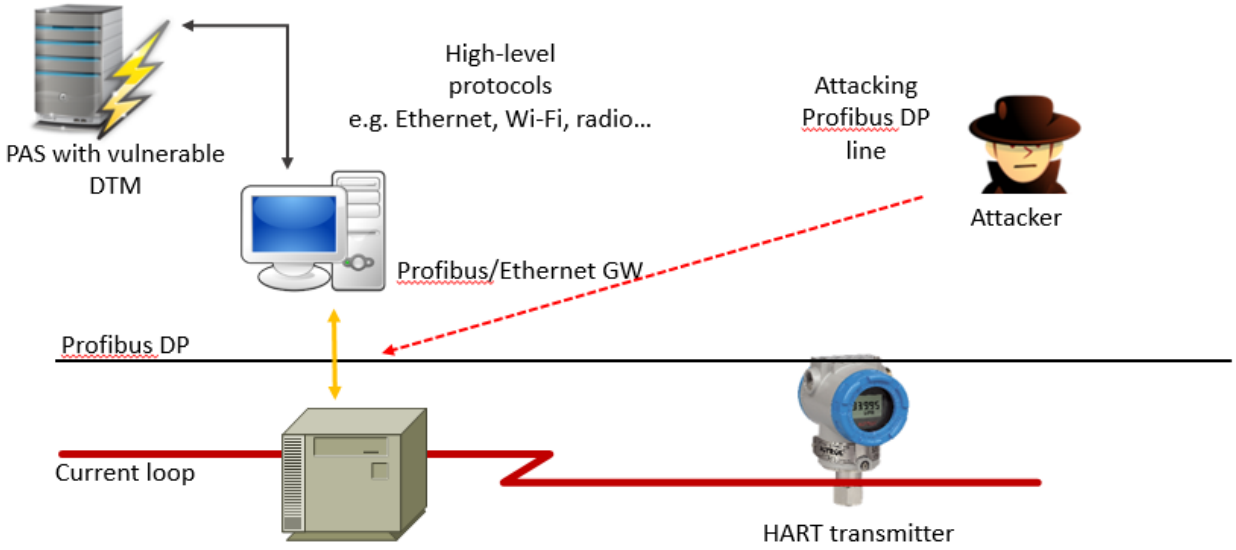

Figure 11: Attacking vulnerable DTM through upper levels (Profibus DP).

Before talking about fuzzing methods, let's see what components we've researched. Unfortunately, we're bound by responsible disclosure, so we can't name the exact device models or exact component names. But we can show the vendor list. We've researched DTM components from the following vendors: ABB, Azbil Corporation, Dresser Masoneilan, Emerson/Rosemount, Endress+Hauser, Flexim, Flowserve, FOXBORO-ECKARDT, GE Oil & Gas, General Monitors, Hans Turck GmbH & Co. KG, Honeywell, ICS GmbH Ettlingen, Invesys/Foxboro, Klay Instruments, KROHNE, MACTek Corporation, Magnetrol, Metso, Pepperl+Fuchs, SAMSON AG, Schneider Electric USA, StoneL, Vega.

All DTMs were 1.2 or 1.2.1 specification versions, most of them (83) were 1.2 version. Interestingly, we've found DTMs that have been created within the last 2-3 years, but according to their internal information, they support FDT specification 1.2, not the 1.2.1.

There are some software developers that specialize in creating DTM components: M&M Software and CodeWrights. They have special helper tools, frameworks and ever autogenerators for DTM component development. In figure 12, you can see the presence of components based on such frameworks in our DTM sample.

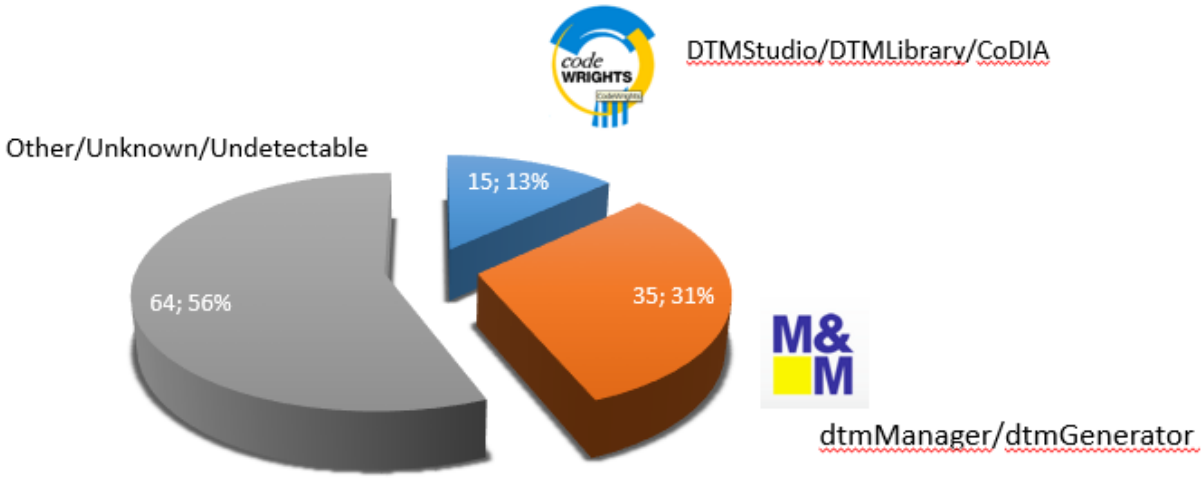

Figure 12: Presence of special frameworks in selected DTMs.

3. Fuzzing tools and methods

After scope definition, we faced the next task – how to fuzz 100 COM components? These components may be written in different languages and use different runtimes, process models, etc. Thus, we've used three different fuzzing methods:

1. Emulate CommDTM and put fuzzed protocol data directly into DeviceDTM (fastest)
2. Emulate device through virtual serial port
3. Emulate device with hardware (HRTshield, ICSCorsair, etc.) (slowest)

If the fuzzing of a component could not be done with method 1, we switched to method 2, etc. Let's review our methods in detail.

The fuzzing infrastructure in all methods is the same: we used the HRTParser lib for parsing and creating HART packets, then a HART emulator (written in Ruby) followed the state of communication between master and slave; lastly, in 2/3 cases, packet internal data went through Radamsa[12]. Target DeviceDTM is located in the FDT Frame container, FieldCare PAS in our case. To trigger the "read all from device" operation, we used special AutoIt[13] scripts.

The method 1 fuzzing schematics are shown in figure 13:

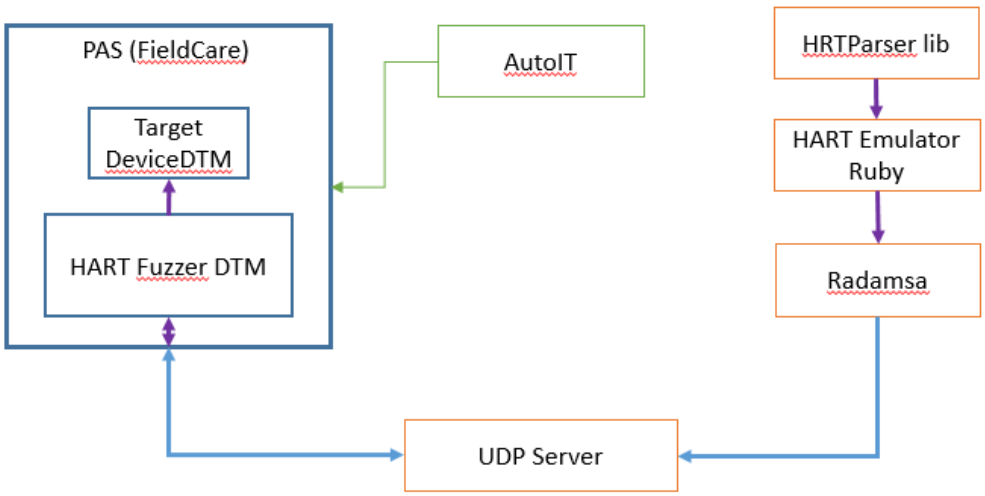

Figure 13: Fuzzing using CommDTM emulation.

[12] https://code.google.com/p/ouspg/wiki/Radamsa, "Radamsa is a test case generator for robustness testing, aka a fuzzer. It can be used to test how well a program can stand malformed and potentially malicious inputs. It operates based on given sample inputs and thus requires minimal effort to set up. The main selling points of radamsa are that it is easy to use, contains several old and new fuzzing algorithms, is easy to script from command line and has already been used to find a slew of bugs in programs that actually matter."

[13] https://www.autoitscript.com/site/autoit/, "AutoIt v3 is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting. It uses a combination of simulated keystrokes, mouse movement and window/control manipulation in order to automate tasks in a way not possible or reliable with other languages (e.g. VBScript and SendKeys). AutoIt is also very small, self-contained and will run on all versions of Windows out-of-the-box with no annoying "runtimes" required!"

We did not want to waste time on low-level communications. HART is slow (1200 baud), and we needed to speed things up. We used a simple scheme: FDT Frame (FieldCare) + Target DeviceDTM + our special HART Fuzzer CommDTM. HART Fuzzer CommDTM is an emulation of a real CommDTM. When the "read from device" operation is started, it takes transaction requests from DeviceDTM, but instead of sending it to the HART modem, it sends it to the fuzzing infrastructure. The Ruby HART emulator script receives query from the UDP server and sends back the HART packet body (without the header). This body is passed through Radamsa in 2 of 3 cases. The AutoIT script triggered FieldCare to repeat the "read from device" operation constantly.

The second method is using virtual serial port emulation and is shown in figure 14:

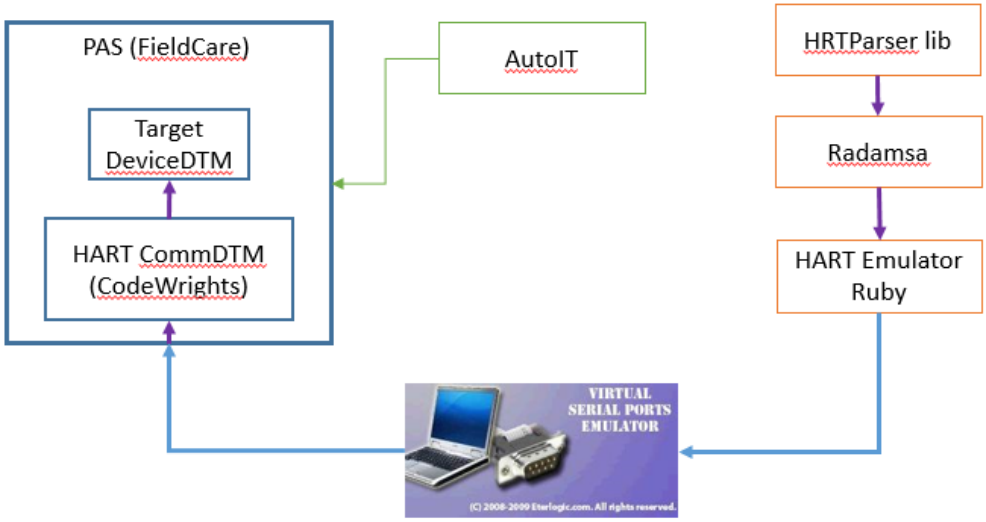

Figure 14: Fuzzing using virtual serial port emulation.

Some DTMs have different thread models or expected HART timings and special operations, so creating a perfect CommDTM simulation could be a bit costly. But we could use a real HART CommDTM (in our case – CodeWrights HART CommDTM) and connect it with Ruby HART Emulator using virtual serial ports emulation software (in our case – VSPE[14]). CommDTM received transaction requests from the target DeviceDTM and sent it over a virtual serial port (thinking that it's a real HART modem connected to the current loop). Emulator takes a packet, strips command from it, uses HRTParser to create the answer. In 2/3 cases, the answer will go through Radamsa. Then, it will be packed in the correct HART packet and sent back over the virtual port.

But in some cases, DeviceDTMs are rather tricky and refuse to work even with a virtual serial port emulation. In these cases, we needed heavy artillery on the battlefield and used the method from figure 15:

[14] http://www.eterlogic.com/Products.VSPE.html, "VSPE is intended to help software engineers and developers to create/debug/test applications that use serial ports. It is able to create various virtual devices to transmit/receive data. Unlike regular serial ports, virtual devices have special capabilities: for example, the same device can be opened more than once by different applications, that can be useful in many cases. With VSPE you are able to share physical serial port data for several applications, expose serial port to local network (via TCP protocol), create virtual serial port device pairs and so on."

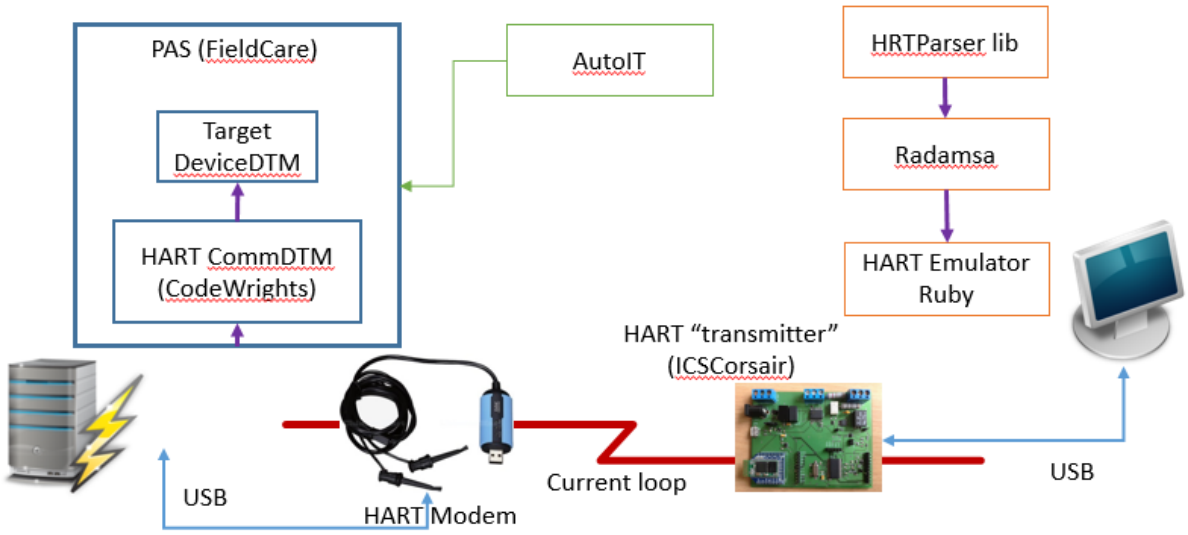

Figure 15: Fuzzing using hardware tools.

CommDTM is connected to the real HART modem, and on the other side, we used HRTShield or ICSCorsair hardware. We could not use a regular HART modem on the fuzzer side because some fuzzing samples could exceed the normal HART packet length of 255 bytes. The regular HART modem can usually receive such packets and pass them further but could refuse to send them. ICSCorsair is configured in the HART modem mode and connected to the fuzzer PC by USB; the Ruby HART emulator used the "serialport" library to work with devices. The other parts of the infrastructure are the same. This fuzzing method reproduces the real environment but is slow and inconvenient.

At the end of the chapter, let's review the tools that we've created for fuzzing.

HRTParser is a Ruby library for creating and parsing HART packet creation. It supports extensions for specific HART devices. The library can be downloaded from http://github.com/Darkkey/HRTParser.

HART DTM Fuzzer is a CommDTM component developed for fuzzing DeviceDTMs. Whenever it receives a TransactionRequest, it will connect to a remote UDP server and request a reply for the command specified in the request. The component is written in C# and compiled for .Net 2.0 and .Net 4.

DTMSpy is a small utility for spying transactions between DTM components. The program is developed by Svetlana Cherkasova using C++.

Ruby HART emulator is a set of Ruby scripts that use **HRTParser** to emulate the HART stack. It can support multiple devices and may be integrated with Radamsa.

HRTShield is a hardware tool: HART communication shield for Arduino boards. We've used a common HART modem IC from Maxim Semiconductors (DS8500) with a passive input filter and an active output filter. The board has a special output signal amplifier. The schematics are available at http://github.com/Darkkey/HRTShield. You can see HRTShield on figure 16:

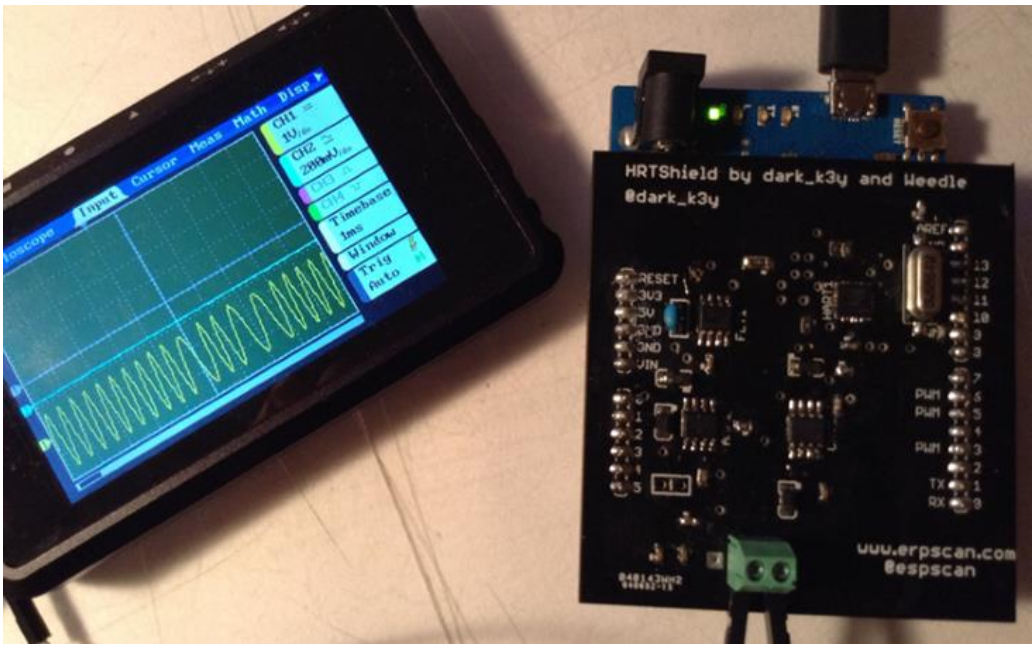

Figure 16: HRTShield – a high-power HART modem.

ICSCorsair[15] is a multipurpose open hardware tool for auditing low-level ICS protocols. It can communicate with various systems using HART FSK, Profibus, and Modbus protocols. You can control ICSCorsair via a USB cable or remotely over Wi-Fi, Bluetooth, or other wireless connections. The schematics, firmware, and software are available at http://github.com/Darkkey/ICSCorsair. You can see ICSCorsair in figure 17:

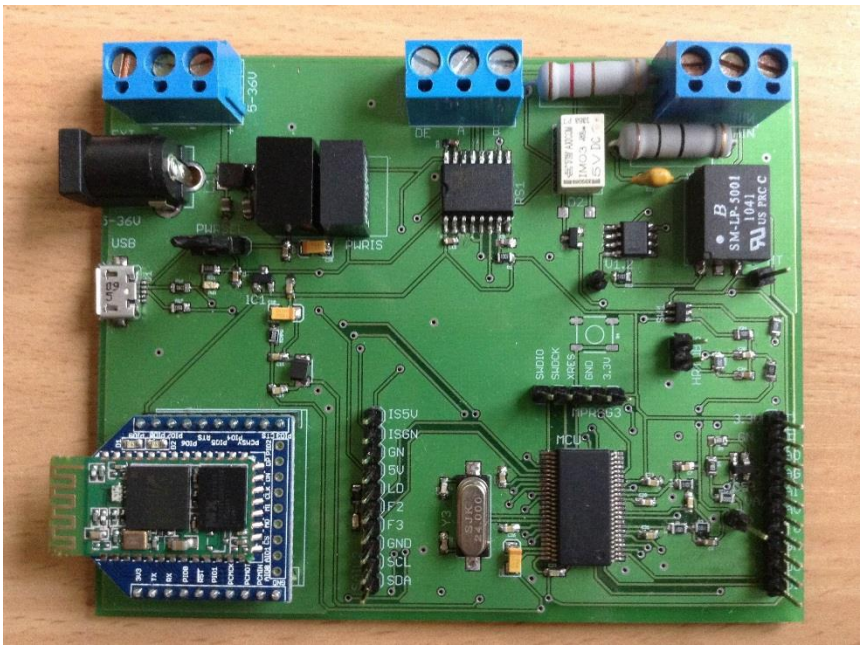

Figure 17: ICSCorsair v.0.03.1 board.

[15] For more information on ICSCorsair, see our BH USA'14 talk: ICSCorsair: How I will PWN your ERP through 4-20mA current loop.

4. Found vulnerabilities and weaknesses

Now, it's time to talk about the found vulnerabilities. Summing up, we've found that 29 components (of 114) are vulnerable. It's about 25%, but if you look at this from the perspective of different device types, you will see a more scary picture: 501 (of 752) device has vulnerable DTM components. This comparison is seen in the figure 18:

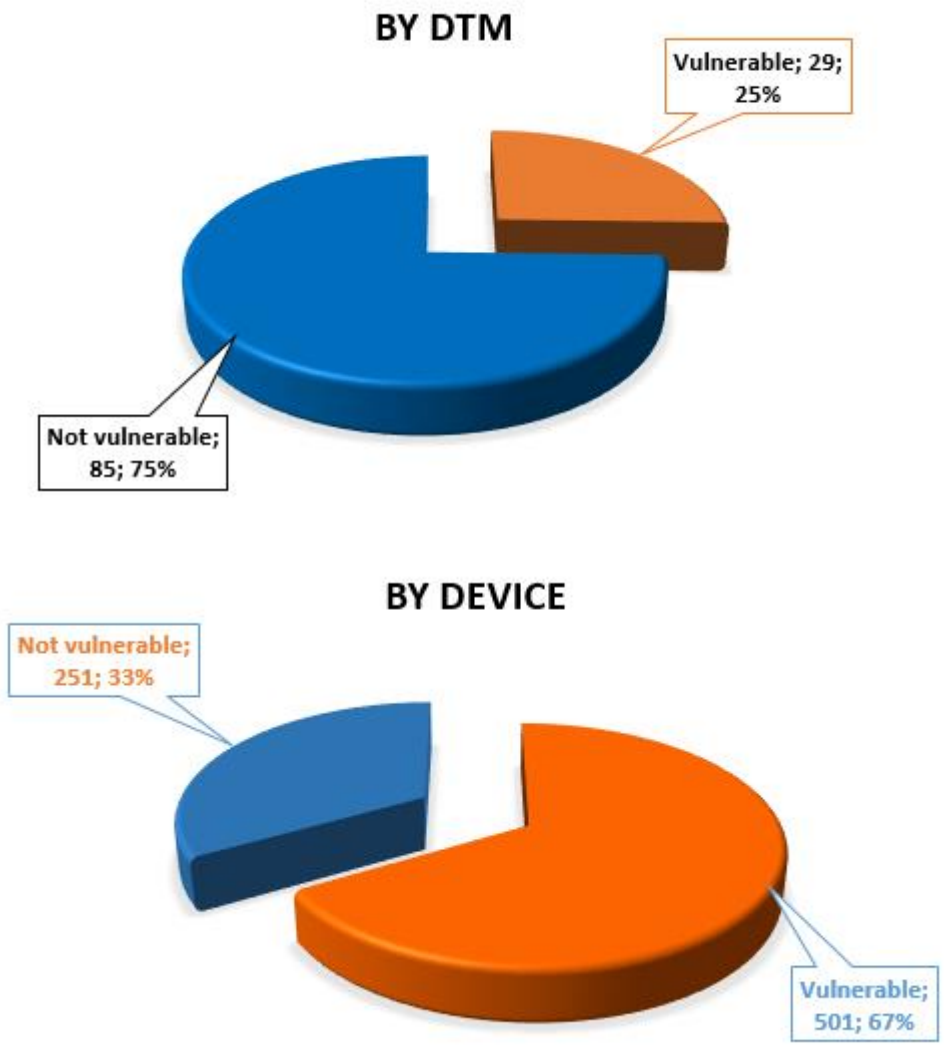

Figure 18: Found vulnerabilities by component and by device.

All found vulnerabilities were divided into six groups:

- Remote Code Execution (RCE) – vulnerabilities that will cause arbitrary code execution in the context of DTM components and Frame applications. This could be buffer overflow or a similar thing. We've written proof-of-concepts exploits for every vulnerability in this group

- Possible RCE – vulnerabilities that could cause arbitrary code execution in the context of DTM components and Frame applications. Writing exploits for heap overflows and similar bugs could take too much time, so we put all vulnerabilities without a stable exploit into this category; this doesn't mean that the vulnerability will not harm a system – in the best case, it's only DoS, and in the worst case, RCE
- Denial of Service (DoS)
- Race Conditions
- XML injections – XML injection in the long tag (or other tag) field could cause an external scheme loading or a Server-Side Request Forgery attack
- Other – we've put strange freezing behaviors, random crashes, crashes on really big packets (>8 kB: these packets can only be sent by HART-IP, and not by HART-over-Current Loop) in this category. They are harder to exploit or cause less harm to the target system. Funny and useless vulnerabilities (like XSS in the autogenerated report view) are also in this category

The vulnerability statistics are shown in the next table:

Vulnerability	Count
RCE	3
Possible RCE	7
DoS	6
XML injection	2
Race Condition	2
Other	9

Table 3: Datalink structure of a HART packet

Distribution of vulnerabilities over vendors is shown in figure 19:

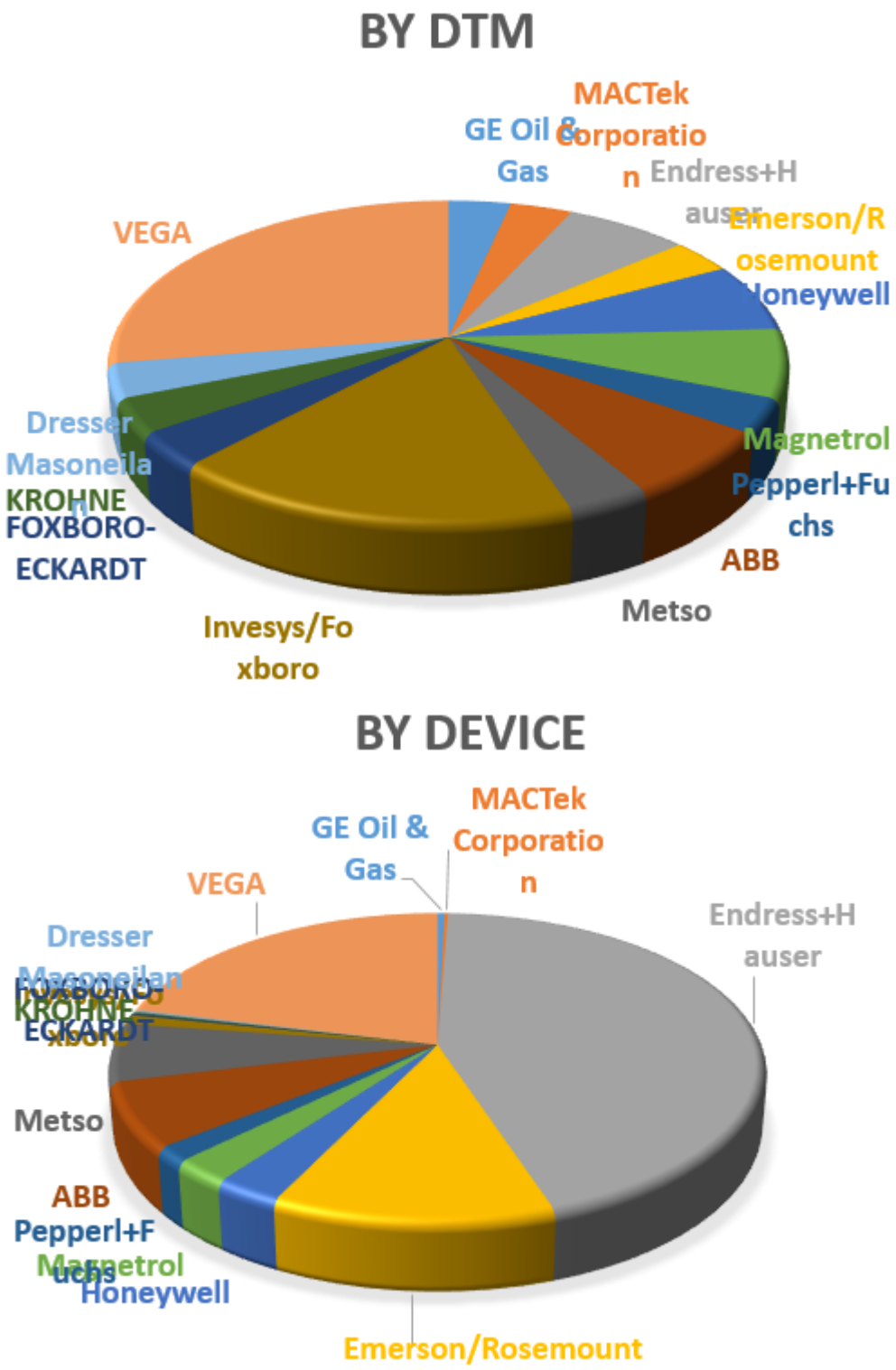

Figure 18: Vulnerability count by vendors.

Another funny statistics is the presence of security mechanisms in DTM components, e.g. stack cookies, DEP and ASLR. Unfortunately, only a limited number of DTMs have all these things enabled. The next table is showing more details on this:

Number of DTMs	Stack Cookies enabled	DEP enabled	ASLR enabled
66	0	0	0
35	1	0	0
5	0	1	0
1	0	1	1
7	1	1	1

Table 4: Security mechanisms in DTM components.

5. FDT 2.0 – is it a solution?

Recently, FDT Group finally introduced a new version of FDT specification, v. 2.0. However, only a few devices support it. The key differences from 1.2.1 are:

- Interfaces are .Net-based
- Class architecture redesigned
- Increased performance
- No XML (the interaction between FDT objects is based on .NET datatypes rather than XML)

These changes look good and should definitely increase the overall security of FDT technologies. However, there are several pitfalls with FDT 2.0:

1. It still has very low spread over the industry; we didn't see any FDT 2.0 components in real industry facilities. Even in the official DTM components catalog (on the FDT Group site), there are no FDT 2.0 DTMs!
2. FDT 2.0 allows backward compatibility with FDT 1.2.1 (using (de)serialization to XML for working with FDT 1.2.*). This could cause problems if used inaccurately
3. Managed code will be not a complete solution if unmanaged code is still used. For example, during our research, we've seen FDT 1.2 components with really secure front-end FDT interfaces, but there was vulnerable managed C++ code inside

Unfortunately, we could not found a real device supported by FDT 2.0 to test it; if you have one, please write to us, and we could borrow it for some time ;)

6. Conclusions and future work

We're planning to actively work with vendors to solve all discovered problems. The full report on the found vulnerabilities will appear (we hope) in the spring/summer of 2015. We could not publish more information at this point because of responsible disclosure. Also, we plan to publish the sources of all used fuzzing tools at the same time.

During our research, we've fuzzed 114 components and found 29 vulnerabilities. If you have one of the 501 different HART devices in your industrial facility, you could be at risk. However, all these attacks are possible not only because of DTMs weaknesses, but also due to fragile ICS architectures. The approach to the whole ICS multilayer networks should be changed. Otherwise, we will face the risks of such vulnerabilities over and over again.

7. Thanksgiving service

This research wouldn't be possible without the help of many people and companies, and we want to mention them. Thanks to:

- **Svetlana Cherkasova && George Nosenko** for "some binary magic" and great help in reverse-engineering and creating proof-of-concept exploits
- **Andrey Abakumov** for help in finding XML injections
- **Fedor Savelyev aka Alouette** for some fuzzing ideas
- **Alexander Popov** for the great background picture