Final report

1. Project details

Project title	"Janus" Industrial Control System VPN Firewall
Project identification (program abbrev. and file)	EUDP17-I: (12560) Janus ICS VPN Firewall
Name of the programme which has funded the project	ForskEL
Project managing company/institution (name and address)	MSB Information Security ApS Karlstrupvej 53 2680 Solrød Strand
Project partners	Fremsyn ApS
CVR (central business register)	36729708
Date for submission	06.07.2020

2. Contents

3. Short description of project objective and results

3.1. Abstract (English)

Secure communication between SCADA systems and the increasing number of distributed smart grid units is a prerequisite for a large renewable power production.
Conventional security systems have developed into large, complex and patchworked systems. Several security breaches have demonstrated that the amount of attack vectors grow with complexity.
The purpose of the JANUS project was to develop a security product similar to a VPN after the "Security-by-Simplicity" concept. The project has successfully developed, tested and demonstrated a security system, that two external independent security experts were unable to compromise. The system has limited requirements for processing power and is well suited for small distributed smart grid units.

3.2. Abstrakt (Dansk)

Sikker kommunikation mellem SCADA systemerne og det stadigt stigende antal distribuerede smart grid enheder er en forudsætning for et elnet med stor vedvarende elproduktion.
Konventionelle sikkerhedssystemer har udviklet sig til store og tunge systemer, der er bygget lag på lag, og en række sikkerhedsbrister har vist, at antallet af cyber angrebsvektorer vokser med komplesiteten.
Formålet med JANUS projektet var at udvikle og demonstrere et VPN lignende produkt til sikker datakommuniktation, som var udviklet efter "Security-by-Simplicity" conceptet.
Projektet har succesfuldt udviklet, testet og demonstreret et sikkerhedssystem, som to uafhængige eksterne sikkerhedskonsulenter ikke kunne kompromittere. Systemet stiller små krav til processering og er velegnet til mindre distribuerede enheder.

4. Executive summary

The JANUS project ran from Q2 2017 to Q2 2020 is a cooperation between the companies MSB Innovation Security and Fremsyn. The purpose of the project was to mitigate some of the mounting cyber security threats against the power system by developing a novel type of a VPN tunnel to protect communication between SCADA systems and distributed smart grid applications.

The product has successfully been developed, tested and demonstrated.

A key in the system was to utilize the "Security-through-simplicity" (STS) concept – making the code as short as possible in order to be able to oversee and comprehend for a human brain. The STS concepts is in opposition to the today widely used de-facto "security-by-obscurity" (SBO) concept, where the code intentionally or by patchwork over time is made complex and intertwined. A number of security breaches over the past years has demonstrated that the complex and intertwined codes increase the number of security vulnerabilities, and the purpose of the STS is to reverse this development.

All objectives of the JANUS project are fulfilled:

- The 'security by simplicity' is implemented in only 1.298 effective code lines.
- A 'defense in depth' concept with several security layers is implemented
- Two external penetration tests by independent experts were unable to compromise the system.
- The VPN product operate at TCP/IP level and does not intervene with protocol levels above TCP. Any communication protocol using TCP/IP can be forwarded by the VPN product.
- The product was successfully demonstrated at the Nerve Smart System BESS product.

The development phases of the project progressed with only minor obstacles and problems, as the lead developer is sublimely experienced in cryptography coding. The project did, however, meet difficulties in securing a suitable demonstration site. In the end the system was successfully demonstrated.

The JANUS VPN concept is successfully demonstrated and it has demonstrated to be very suitable for small distributed smart grid applications, as it combines a high security level with low processing requirements.

Several companies that could be potential customers and/or partner for further development have expressed interest in the product. The product is ready to move on to an actual commercial implementation when the economic activities pick up speed after the conora times.

5. Project objectives

The objective of the project is to proof the concept, develop and demonstrate a novel information security product for protecting Industrial Control Systems (ICS), e.g. SCADA systems, against cyber-attacks.

Main features of product:

- Utilizing a "security by simplicity" concept with a protocol implementable in about 25.000 lines.
- utilizing a "defence in depth" concept with several security layers
- compatible existing IEC standards in the energy sector

Project work on project was divided into 5 Work Packages.

6. Original project plan

6.1. WP0: Project management

Purpose	Coordination and general product management. Contact and reporting to ForskEL.
Activities	Project coordination. Active contact with Advisory Board incl. quarterly meetings and news mails. ForskEL contact and reporting. Final ForskEL reporting.
Deliveries	Quarterly project updates. Final reporting to ForskEL.
Milestones	1. Final ForskEL reporting completed
Resources	500 h Project Manager.
Risks	No major risks identified.
Responsible	FREMSYN

6.2. WP1: Software foundation

Purpose	Write software foundation to facilitate a network-connected daemon process without the need for any middleware. This software foundation should provide the lower software- based layers of defence in depth.
Activities	Write software. Basic software utilities, I/O, basic cryptography, daemon interaction. The software foundation should be implemented with focus on simplicity, robustness (hard to misunderstand and misuse, misuse should not compromise system) and ease of use. The code should have a decent unit test coverage. The WP includes setting up infrastructure for automated build and testing ("continuous integration").
Deliveries	Software foundation to facilitate subsequent WPs.
Milestones	1. Automated test systems configured and operational

	2. Basic libraries implemented 3. All libraries implemented
Resources	1600 h Senior Software Developer. 400 h Junior Software Developer.
Risks	The team have considerable experience from similar projects, and the technology risk (that it is impossible to create the software) is low.
Responsible	MSB Information Security

6.3. WP2: Design and implement communication- and security protocols

Purpose	Design and implement communication- and security protocols as well as external review of those based on the frameworks implemented in WP1.
Activities	The protocol should be designed and implemented with simplicity and robustness in mind. As a consequence, it is likely that only symmetric cryptographic can be used.
Deliveries	Verified implementation of communication- and security protocols.
Milestones	1. Protocol designs documented. 2. Protocol designs reviewed. 3. Protocols implemented. 4. Protocol implementations reviewed.
Resources	900 h Senior Software Developer / Senior Cryptographer. 150 h Junior Software Developer. External review.
Risks	As the team have high experience from similar projects, the technology risk is low.
Responsible	MSB Information Security ApS

6.4. WP3: Demonstrate VPN - and firewall device

Purpose	Implement, test and demonstrate VPN- and firewall device.
Activities	A test host will be found, preferably in cooperation with large Danish power utilities. The product will be implemented and tested. Implement a working VPN and firewall device base on the framework of WP1 and the protocols on WP2. The scope of the implementation is to verify, test and demonstrate a working product.
Deliveries	Test implementation. Test result. A software daemon implementing the basic features of the 'Janus' VPN firewall.
Milestones	1. Implementation at test host 2. Successful test run 3. A working product suitable for demonstration.
Resources	600 h Senior Developer. 200 h Junior Developer.
Risks	As the team have high experience from similar projects, the technology risk is low.
Responsible	MSB Information Security ApS

6.5. WP4: Communication and marketing

Purpose	An aggressive dissemination and communication strategy will be pursued to 1) ensure public scrutiny of the project and 2) raise awareness among potential customers of the product. Establish the vision and product in the market.
Activities	Participation at international cryptographer conferences Articles in relevant media and participation at national seminars in the energy sector Create webpage and sell complementing consultancy services to gain market insights. Elaborate on- and mature visions and

	product presentation material. Write, submit and present talks and papers. Identify and initiate contact with potential initial customers.
Deliveries	1. Webpage 2. Final go to market strategy 3. Articles for relevant media 4. Awareness and market presence.
Milestones	1. Presentation at 2 international cryptographer conferences 2. Direct marketing at all major Danish power utilities
Resources	100 h Senior Cryptographer / Senior Software Developer 650 h Project Manager (Business Developer)
Risks	Lack of interest from the industry. Mitigated by broad network. No significant risks identified. The team has the needed experience.
Responsible	FREMSYN IVS

7. Actual project execution

The implementation of the project has followed the 5 working packages of the project closely.

Highlights of the project:

- The project was awarded in March 2017.
- Work on the automated software platform for build and test started immediately after official project start in Q2 2017. During 2017 the software bench for test and development was rapidly expanded with functionalities and various test methods.
- Core libraries containing basic functions and support functionalities were developed during Q1 and Q2 2018.
- An Advisory Board meeting was held at 2 February 2018 with main focus on developments, products and business models.
- A number of end-user interviews were held in spring 2018 to narrow in focus for product development.
- It was concluded based on Advisory Board meeting and the end-user interviews that the project should focus on development targeted at small household grid connected units like household meters. The Janus concepts has a strong competitive advantage in requiring low computing power.
- The project had a very interesting dialogue with Erik Andersen from Q2 2018. Erik is a core specialist in internet protocols for the energy sector with key knowledge for the project. His contact was supplied through the advisory board.
- First version of the project webpage was launched at Q2 2018.
- The project participated with an exhibition at the "Homeland Security Conference" in Copenhagen at 15 November 2018.
- Functionalities and testing continued during autumn 2018. System ready for first QA at this stage, and first operational version of the software was ready by early 2018.
- An external vulnerability test (software test) of the Janus concept was planned and performed during 2019. Test results were very promising.
- During 2019 a lot of efforts were spent on finding a suitable locating for test at a hardware system. This proved to be a significant challenge, as there had recently been a lot of focus on security and no one was interested in opening their systems to the point necessary for testing. Many actors were approached without success.
- Finally, an agreement was made with Nerve Smart Systems that participate with Fremsyn in several projects (Horizon2020: Insulae and EUDP: TOPChargE) setting up a local DC grid with BESS capabilities at Bornholm.
- Hardware tests were performed in autumn 2019 together with Nerve Smart System. The test demonstrated that the system works as planned.
- Final reporting concluded in spring 2020.

7.1. Progress of WP0 Project management

The purpose of this WP was to secure an efficient project management and to coordinate contact with ForskeEL/EUDP and the Advisory Board. The tasks of the project meeting progressed as planned in the beginning of the project.

- Quarterly economic overview has been made for EUDP in accordance with schedules.
- Annual reports of the progress have been made in accordance with project schedules.
- An Advisory Board meeting was held early in the project in February 2018. The Advisory Board has since this time been used for counselling and contacts in the business.
- Final reporting is concluded with a short delay.

In generel we regard the progress of WP0 as a success.

7.2. Progress of WP1 Software Foundation

The purpose of this WP was to develop the basic software foundation for the Janus concept. The work package progressed almost punctually as planned in the project planning period. This was also expected due to the immense experience from the lead programmer.

- Work started immediately after official project start.
- Basic functionalities and test bench developed throughout 2017 and 2018.
- Scope concluded by autumn 2018.

The planning and execution of this work package was very successful.

7.3. Progress of WP2 Design and implementation

The purpose of this WP was to design and implement the security protocols of the Janus concept. The work package developed as planned.

- Dialogue with Erik Andersen from Q2 2018.
- First version ready for operations and QA in early 2018.
- Continued development, testing and implementation throughout 2018 and 2019.
- External vulnerability report carried out late 2019.

All WP objectives were successfully implemented and demonstrated.

7.4. Progress of WP3 Demonstrate VPN

The purpose of this WP was to demonstrate and proof the Janus concept at an actual hardware system. In the planning phase it was planned to carry out the test in cooperation with a Danish utility but we were unable to find a partner interested in a test. Instead we ended up by implementing it at a battery system in cooperation with Nerve Smart System.

- Dialogue with utilities on test and cooperation possibilities throughout 2018. An interested test partner was not located.
- Change of focus from utilities to distributed smart grid devices. Dialogue with potential partners throughout 2018 and 2019.
- Test with Nerve Smart System end 2019 with proof of concept.

In generel this work package took a lot more effort than anticipated but we succeeded in the end to demonstrate and proof the Janus concept.

7.5. Progress of WP4 Communication and marketing

The purpose of this WP was to ensure external communication with potential business- or cooperation partners and with the general public in broad.

- Project website from Q2 2018.
- Contact with various end customers Q3 2017 – Q3 2018 to collect design input. Design inputs were discussed with contacts in utilities, equipment manufacturers and distribution companies related to the Danish energy sector.
- Homeland Security Conference Q3 2018.

- Discussions with contact on test site and business cooperation throughout 2018 and 2019.

In generel this work package took a lot more effort than anticipated at the planning stage of the project. We ended up securing 1) a test site, and 2) two different partners interested in actual implementations of the project concept, but it took a lot of effort. This was the crucial success parameters which was then achieved.

8. Risk assessment

8.1. Analysed risks before start of project

- The technology risk is considered low as the involved technologies are within the comfort zone of the consortium members.
- There is a commercial risk as the market is dominated by very large and powerful vendors.
- Lack of commercial interest from customers is a risk, but it is very low as the project consortia has experienced a very significant preliminary market interest. Risk mitigated by sales experience and network.
- IP rights and patent conflicts with big players is a risk, but the security by simplicity approach is novel in the ICS branch and the risk is perceived low.
- Personel risks of key personel leaving the project is probably the most significant risk. This is mitigated by the extensive network of the participants in the relevant branches.
- Financial risks are mainly depending on the side activities in the participating companies. Both companies have significant customer portfolios and the risk is not perceived as significant (given ForskEL funding).

8.2. Risks discovered during the project

- The personnel risk turned out to be more severe than initially anticipated. We did not manage to find additional personnel with the right qualification in cryptography.
- The energy industry saw a lot of cyberattacks during the first year of the project. Some were due to the unfolding conflict in Ukraine, some due to ransomware attacks in Denmark. The increased amount of attacks heightened the threshold requirements in the IT security business – but potentially it has also increased the value of the project.
- The increased demand for security made it hard to find a test site. In the end a test partner was found, but it took long time and delayed the project significantly.
- Towards the end of the project, the corona crisis and its ensuring unfolding economic crisis have put an interested external customer at hold.

9. The project

9.1. The problem

The main problem addressed by this product is that most computer software has grown extremely complex. This complexity will unavoidably lead to software errors. Some of these errors are exploitable by cyberattacks to grant uninvited third parties' access to the infrastructure hosting the software and thereby access to sensitive systems and confidential information.

Main Technical Problem to Solve:
Software Complexity -> Software Errors -> Cyber Attacks

Many examples of this problem have come to the public’s knowledge recent years – and there are probably a lot more which have not been published by the cyber criminals since that would allow the software errors to be corrected and thereby the value of the knowledge of the errors to vanish.

A few of such examples are (see links in footnote for further examples):

- **The Shadow Brokers:** A group calling themselves "The Shadow Brokers" have August 13th 2016 published examples of attack code allegedly originating from NSA (National Security Agency in USA). The code includes among others attacks on Cisco firewalls and Fortinet firewalls. This example illustrates how resourceful organizations (e.g. state sponsored) possesses vulnerability knowledge and very strong technology which allows then to bypass many well-established security technologies and products.
- **Stuxnet:** An advanced work targeting SCADA systems and capable of infecting even strongly protected infrastructure. It is believed to be developed by USA or Israel to target Iranian uranium centrifuges.
- **Heartbleed:** A vulnerability was announced in the OpenSSL communication encryption library in 2014. This vulnerability potentially allows an attacker to read from RAM in affected servers. This could (at least in theory) e.g. be used to read cryptographic keys whereby communication as well as the server's identity could be compromised. This vulnerability was particularly severe since most internet-facing security-related servers on the Internet uses this library.
- **CVE-2019-14899:** Vulnerability that existed on most Linux distros allowing a network attacker to interfere and hijack VPN connection.
- **Cisco VPN vulnerability:** at 2018 the critical vulnerability was discovered in webvpn feature affecting the Cisco's line of Adaptive Security Appliance products. Vulnerability could allow an unauthenticated, remote attacker to cause a reload of the affected system or to remotely execute code.
- **Meltdown & Spectre:** critical vulnerabilities in modern processors. Potential attack misuse features for higher performance such as Speculative execution, Cache, Out-of-order execution and vulnerabilities in processors firmware. Meltdown allows a program to access the memory, and thus also the secrets, of other programs and the operating system. Spectre allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets.

9.2. The vision

Security through simplicity

Most communication protocols (and IT solutions in general) suffer from extreme complexity. To implement most modern communication protocols – including for example the IEC 61850 suite – requires a significant number of software modules with several millions of program code lines. This is a problem since there is a correlation between a software program's size in code lines and the number of software code errors: The more code lines, the more errors are to be expected.

Software code errors in communication protocols are very problematic, as these potentially can be used by adversaries to attack the software and thereby the host systems running the software and thereby the networks and the infrastructure in which these host systems reside.

Our vision is to define a security protocol, which is implementable in about 25'000 lines of code. This is not only drastically less than existing protocols (whereby a drastic reduction of errors can be expected) but even more important, sufficiently small to allow one human to understand and keep an overview of the entire software and to allow for efficient and complete review and quality control.

It is important to underline that this simplified communication protocol is not intended to replace existing communication protocols, e.g. IEC 61850, but only to facilitate and protect communication between networks. The actual devices on these networks will communicate using the "ordinary" protocols and not know about this security product's presence in the network.

Defense in depth

A central security concept to be used in the development of the 'Janus' VPN Firewall is 'Defense in Depth'. In this concept, the product is built with several layers, which – ideally – each are strong enough to resist attacks by themselves. In the software implementation, such layers will be identified early in the design phase and a constant attention will be paid on securing these layers during the implementation.

Prevent, contain, detect, restore

Typically, strong IT security must contain the following four components:

- **Prevent:** Prevent the attacker from exploiting your infrastructure in the first place. For example, by firewalls, anti-virus, and other security appliances.
- **Contain:** Ensure that if you are compromised, then the attacker cannot access your entire infrastructure but only a subset thereof. For example, by segmenting networks with strong firewalls in between.
- **Detect:** Ensure that you can detect if your infrastructure has been compromised. For example, by monitoring network traffic or which programs are running on your systems.
- **Restore:** Ensure that you are capable to restoring your system to a clean and operational state after an attack. For example, restore of backup or re-installation.

This product targets the first two components, the proactive *Prevent* and *Contain*. It aims to prevent attacks in the first place as well as prevent spreading of infection by providing a solution which, ideally, does not contain software errors that can be exploited by an attacker to mount an attack that can provide access to the protected network(s) or infrastructure.

9.3. Objective

The objective of the project is to develop and demonstrate a novel information security product for protecting Industrial Control Systems (ICS), e.g. SCADA systems, against cyber-attacks.

The product will:

- utilizing a 'security by simplicity' concept with a protocol implementable in about 25.000 lines.
- utilizing a 'defense in depth' concept with several security layers
- be compatible existing IEC standards in the energy sector

The content of this project will proof the concept and develop the product till a state where it can be demonstrated.

9.4. Choise of programming language and platform

One of the most essential design decisions is the choice of programming language and platform. This choice defined almost all subsequent work.

Overall, programming languages can be categorized in multiple dimensions as listed in the following sections. Some of these dimensions are not completely independent as the number of programming languages is finite and since each programming language has a unique position on most dimensions.

9.4.1. Dimension 1 – Type of programming language

There are two main classes of programming paradigms: Declarative and Imperative.

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. Programmer merely declares properties of the desired result, but not how to compute it. A widely use example of declarative language is SQL.

Imperative programming is a programming paradigm that uses statements that change a program's state. Programmer instructs the machine how to change its state. Imperative languages could be subdivided into:

- Procedural: instructions are grouped into the procedures/functions
- Object-oriented: instructions can be grouped together with state they operate on

Modern computer processors run imperative code and they provide hardware support for procedural code. Hardware support for other types of programming is possible but attempts have not been commercially successful. Other types of programming are usually run in a runtime environment and their instructions are translated into imperative machine code executed by processor.

Imperative language was chosen due to its closeness to hardware innerworkings and possibility to compile it and execute it with minimal amount layers of abstraction which could contain error and additionally keeping program small end efficient. A combination of procedural and object-oriented programming will be used as these two can co-exist and since object-oriented programming can be used to obtain further abstraction (i.e. encapsulation) which can be used to reduce complexity and risk of errors.

9.4.2. Dimension 2 – Abstraction level

There are several abstraction levels to choose from:

Machine languages The CPU instructions in their native (i.e. binary) form. Extremely hard to use, no advantages compared to assembly language. Should never be used.

Assembly languages: Human-readable text commands representing the processor's machine language instructions on a 1:1 basis. Compiled into a machine language for execution. The code is very large in size and lacks most modern programming language features. Very expensive and risky to use due to its size and due to being hard to understand by humans.

Compiled languages (to native code): compiled languages are converted directly into machine code executable format (binary file) that the processor can execute. They tend to be faster and more efficient to execute than following types. Abstract enough to allow automation, abstract concepts and data structures. Provide high level control over hardware they run on.

Compiled languages (to bytecode): Bytecode consist of numeric codes which are form of instruction set designed for efficient execution by software interpreter. Bytecode is executed by runtime environment and interpreted into the machine code. They are slightly slower than languages compiled to native code. They have easier cross-platform interoperability.

Interpreted languages: require interpreter which run through program (usually every time it is executed) line by line and interpret it machine code. In general, less performance efficient than compiled languages but easier to learn and program in. They are separated from hardware by many layers of abstraction – its interpreter environment.

We decided to develop choose language compiled to native code because if its performance, efficiency and simplest run-time environment. Also, most modern programming language features are supported by languages compiled to native code.

9.4.3. Dimension 3 – Memory management

Managing memory is an essential problem for applications. There are two main strategies for memory management: explicit memory management and garbage collection.

Explicit memory management is a strategy where the application is responsible for allocating its memory as well as explicitly freeing its memory when not in use anymore.

Garbage collection is an alternative strategy where the application allocates the memory and objects it needs, but does not bother with freeing it again. The garbage collection system will automatically determine when a memory segment or object is no longer being referred and will then automatically free it.

Explicit memory management has historically been the cause for many vulnerabilities in software. Typically, due to use-after-free or due to buffer overrun errors. This type of errors is typically not possible when using garbage collection for memory management. Garbage collection also has issues, including non-deterministic destruction of objects, the complexity of the garbage collector framework and performance.

We have decided to use explicit memory management – partly since garbage collection by itself adds a considerable complexity, and partly since it is only supported by languages and frameworks.

To overcome the security risks related to explicit memory management, a special software framework will be constructed which encapsulates the memory management routines and prevents buffer overrun errors and makes use-after-free highly unlikely.

9.4.4. Dimension 4 – Supporting libraries and middleware

The software that is developed will often co-exist with other software. One – significant – other component is the operating system, which is discussed in the next section. But other external software often includes various supporting libraries and middleware.

To be true to our vision of minimal complexity we should keep the amount of external code via supporting libraries and middleware as low as possible. However, some essential supporting libraries can probably not be avoided, such as the standard C libraries used to communicate with the operating system.

9.4.5. Dimension 5 – Operation Systems

In industry prevailed two families of operation systems Windows and Linux.

Microsoft Windows is group of proprietary operation systems develop by Microsoft including Windows 10 and Windows Server. It is popular operation system for PC and laptops. It is also used in many IOT systems.

Linux is a family of open source Unix-like operating system based on the Linux kernel. There are many (hundreds) Linux distributions developed by various companies, organisations and developers suitable for almost any imaginable use case. It is operation system of choice for internet infrastructure and small, power and cost-efficient embedded systems including many IOT systems.

No operating system is an alternative – in the sense that the features of an operating systems is embedded into a monolithic application. For this to be realistic, the underlying hardware platform must be very simple and the same goes for the communication protocols (e.g. to implement the TCP/IP protocol and LAN network drivers is not realistic; but a serial-based protocol on a micro controller may be realistic).

Project Janus supports both operation systems in both 32-bit and 64-bit versions and it is designed to be portable on other platform in the future.

No operating systems may become relevant for special future applications, e.g. for some types of small IoT devices.

9.4.6. Dimension 6 – Developing complexity

Developing complexity is an important parameter. If the complexity is too high, this could by itself become a security issue (too hard to program -> higher risk of errors). Also, complexity affects programming efficiency (how long time it takes to implement a given feature) and the availability of competent developers.

Complexity should be assessed from a holistic perspective, i.e. it should not only focus on programming language but also related tools, development environment (IDE), etc. as well as middleware and external software frameworks.

Related to complexity is also the discipline needed to use the platform and to what extend the platform "nudges" appropriate discipline.

9.4.7. Dimension 7 – Maturity

The maturity of the programming language and associated tools, middleware, libraries, platform, etc. is essential. If they are immature, the risk of errors affecting our software is too high. Relating to this is also the future availability of support and maintenance for the components.

9.4.8. Dimension 8 – Hardware platform

Hardware platform has not been a parameter for our choise of programming language and platform. But never the less, the hardware platform constitutes a significant part of the complexity of a complete product.

A less complex hardware platform is desireable for several reason. Obviously for security – many vulnerabilities have been identified in hardware platforms which are related to too high complexity. Furthermore, more complex hardware platforms typically also lead to higher costs and higher power consumption.

The choice of software and hardware platforms are related since they have to support each other.

9.4.9. Chosen language and platform

C++ was chosen as programming language.

The language itself and the tools related to it (compilers, IDEs, etc.) are highly mature. The language has been a very frequent used programming language for decades.

To overcome the inherent risk related to memory management, a special software framework will be constructed which encapsulates the memory management routines and prevents buffer overrun errors and makes use-after-free highly unlikely.

Standard tools (compilers, IDEs, build frameworks, etc.) will be used. Essential C libraries for OS interaction will be used.

9.5. Common sources of complexity

A software program often consists of a one or more of the following main components:

- **Application Logic**
  The application logic implements the main functionality of the software, such as how to reach to various situations and values.
- **Communication and Persistation**
  Communication and persistation is responsible for communicating with other components and for storing information for later use.
- **User interface**
  The user interface is responsible for communicating with the software's operators.
- **Cryptography**
  Cryptographic libraries typically has a role of supporting character. Nevertheless, they often add a significant contribution to the overall complexity.

Each of these have their own properties with respect to complexity, which are discussed in the following sections.

9.5.1. Application Logic

Since the application logic is processing all the information and is responsible for taking the decisions, this is typically also the most relevant part to protect. The application logic is often the smallest part of a software product (when considering the total code; and often the one where existing sw cannot be used). Since the application logic often does not include large external software libraries, it is typically not the primary source of complexity.

9.5.2. Communication and persistation

Protocols for communication and/or persistation have become quite complex and adds up to millions of lines of code in most applications. The primary reason for the complexity of the software implementing the protocols is that the protocols themselves have become very complex. A part of the reason for the complexity is that many protocols are built on many layers of abstraction. This abstraction – if done wrong – can be double bad in the sense that they increase the overall complexity and furthermore facilitates that attackers can misuse functionality to mount attacks or hide attack activity (for example many of the features of XML files, such as character encoding). This abstraction and complexity are often not strictly needed for the core functionality of a protocol or application.

An important note in this context is that the protocols for communication and/or persistation are frequent targets of attacks since they are an application's interface to networks (and data stored on disk) which are typically from where attackers enter the systems they attack.

One challenge with protocols for communication and persistation is compatibility with existing systems and standards. In order to stay compatible, protocols may not be changeable.

9.5.3. User interface

User interfaces can be a substration source of complexity. In particular if the user interface relies on advanced graphics. From a security perspective, the user interface should receive appropriate attention in order to ensure that e.g. attacks taking place on the end-user's computer (e.g. attacks attacking the content of a web browser) are handled and to ensure that appropriate input validation is in place.

Making the user interface itself simpler can often make the software implementing the user interface simpler. Also avoiding the most advanced graphical features may reduce complexity significantly without too high cost from a user-friendliness and/or user-experience perspective.

9.5.4. Cryptography

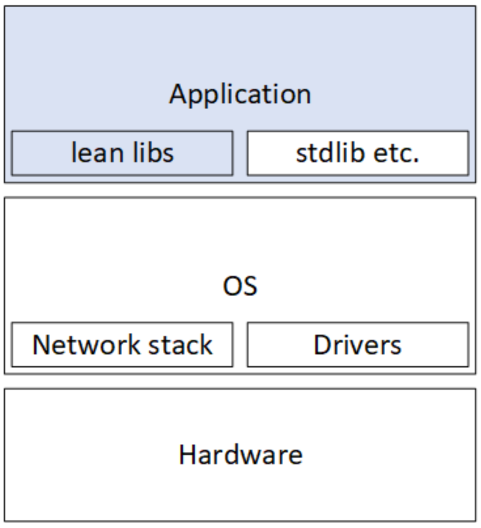
Cryptography does in some applications account for a substantial part of the total complexity. This is in part due to the inherent complexity of some of the algorithms and protocols (in particular the asymmetric algorithms and protocols) and in part due to some of the protocols and formats (i.e. persistation as in a previous section) are rather complex. This is further worsened by the fact that many cryptographic protocols have a significant legacy payload (being able to communicate with older versions or differently configured protocols).

9.6. Segmenting the software

It is in many situations an advantage to segment software. By doing so, each part can stay small enough to be humanly understandable. Furthermore, it may allow different segments to have different security levels. For example, the user interface may be handled by a separate process isolated from the business logic. A positive side effect of this approach is that a vulnerability in one segment probably will not expose another segment. One negative side effect is that the total complexity of the combined application will typically be higher compared to a more homogeneous application. Another example of separation could be to implement some communication protocols in separate processes, in particular if these protocols have a "secondary" nature, such as rarely used backwards compatibility.

9.7. The resulting technology stack

The technology stack is illustrated in this figure:



The application and "lean libs" are implemented according the paradigms of this work (marked with background colour). The remaining parts are existing software and hardware platforms.

"lean libs" are the function-specific libraries implemented as part of this work. Function-specific libraries exists for general memory management, for IO (disk and network), for symmetric cryptography, and for parallel computing (threads, critical sections, etc.).

The technology stack is also important to keep in mind with respect to how much complexity is not in the hands of this project.

9.8. Defence in depth

Defence in depth – i.e. to have multiple layers of security that (ideally) all have to be compromised to attack a system – is an important concept for building a secure application. The layers should each be designed with robustness in mind and thus that a compromise in one layer does not automatically escalate to other layers.

Defence in depth in "lean libs"
"Lean libs" are the general libraries implemented to handle shared functionality. To facilitate security in depth, they are implemented with the following layers:

1. **Memory handling**
   Allocation, reallocation and freeing of memory only takes place in one class. Each system call is only called from one place in the entire codebase. This allows for thorough checks to be performed related to these calls and it allows for the code itself to be thoroughly reviewed. All access to memory blocks (except for accessing buffer pointers for a few system calls related to e.g. I/O) are via methods that checks for buffer overrun before performing the action.
   A strict regime for handling objects has been developed, which includes guidelines for how has ownership of objects and for RAII (Resource Allocation Is Initialization) patterns.
2. **OS interaction**
   All OS interaction (e.g. for I/O) is isolated into its own function-specific classes and kept a compact as possible and only with essential features. This allows for thorough checks, review, and error handling.
3. **Communication and persistation**
   An automated code-generation framework for generating persistation objects is being written. This allows the majority of the
4. **Error handling**
   All error handling in the application is via exceptions. This ensures that an error situation does not go un-noticed and, at the same time, makes error handling less complex.

Defence in depth in the application
The application implements the specific functionality. The application should be written with these guidelines in mind:

1. **Defensive code**
   Try to predict errors and unexpected situations/states and make sure to be able to handle them. In particular when communicating with other systems, it is important to handle unexpected or undocumented situations in a safe manner.
   Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety or security is needed. One of the measures taken was to disable copy and assignment constructor on defined objects because there are states in which they are not save to copy.
2. **Fail fast**
   If e.g. a variable has an illegal state or value, the program should respond to this and fail as early as possible. It is better to stop normal operation rather than attempt to continue a possibly flawed process.
3. **Server-side input validation**
   Web-based applications should perform validations of input on server-side and assume that the client/browser cannot be trusted.

Defence in depth in the underlying operating system and hardware platform
When running a program on an existing OS / infrastructure, the host system should

4. **Firewall**
   A restrictive firewall only allowing essential protocols (in and out) is essential.
5. **Remove unused components**
   Only run essential software/services/daemons. Make sure that no other components are running, installed or available.
6. **Block debug channels**
   Block all debug channels when the system is running in production. In particular on

hardware level, debug channels can be a very helpful tool to an attacked. If debugging is strictly needed, use e.g. an encrypted file to store the information in order to make sure that the information is only available to the intended audience.

7. **User- and rights management**
   It is essential to ensure that strong passwords etc. are used. Likewise, it is also essential to ensure that applications are run as a user that only has the rights strictly needed by the software's functionality.

An example of an attack path that spans several of the layers listed above: An attacker exploits a vulnerability in a communication protocol. This vulnerability is used to put too much data in a buffer (i.e. exploit a buffer overflow error) which in turn leads to a system call (due to data written to other objects, perhaps even on the stack) with "wrong" arguments. Since the software is running with more privileges than strictly needed, this system call leads to external software being installed which can communicate to external parties not blocked by outgoing firewall rules. By implementing defence in depth, each step should ideally be prevented in its defence layer. But even if a few steps were not prevented (for whatever reason) the attack would we stopped at other defence layers.

9.9. Quality assurance

QA (Quality Assurance) is a set of activities for ensuring quality in software engineering processes. It includes process in whole development cycle and define standards to ensure that the product is designed and implemented with correct procedures.

Rigid compiler settings: all languages went through evolution and often standardisation process was performed after first few versions of language. By default, many compilers do not enforce later and stricter standards. The strict compiler mode warn developer about potential issues that might otherwise get unnoticed in build noise and ensure that code follow the desired standard.

Automated build and QA: Automated build and robust testing environment including unit tests, integration tests, and memory tests. Unit testing are performed on smallest testable parts, testing all individual components of software and check if they behave as expected. In integration test individual units are combined and tested as a group. Its purpose is it to expose defects in the interfaces and interactions between components.

Dynamic testing: Automated dynamic testing is furthermore performance as part of the automated build and QA. Memory tests are performed using Valgrind to check correct functioning of memory management. Concurrency tests are performance using Valgrind to check if the concurrency management framework works as intended and if it correctly used in the application.

9.10. Cryptography

The Janus code base does at this point of time only support symmetric cryptography (i.e. AES for encryption, SHA2 for hashing, HMAC-SHA2 for massage authentication.

Not to implement asymmetric cryptography was an intentional decision since current standards for certificate and PKI infrastructure is very complex (both the protocols in general, their serialization/file formats, and the cryptographic primitives).

Symmetric is much simpler and more robust, but also limited in functionality.

Since some cryptographic properties and features cannot be implemented without asymmetric cryptography, we probably have to implement it at some point of time in the future. However, some formats and protocols may be implemented using the automated code generation framework as mentioned in section 9.18.

9.11. VPN product

A VPN client and VPN server has been implemented. The VPN product operates at TCP/IP level and does not intervene with the protocol levels above TCP. The VPN product facilitates secure communication between two systems and – due to its simplicity – makes vulnerabilities like the one found in the Cisco VPN products (see section 9.1) much less likely.

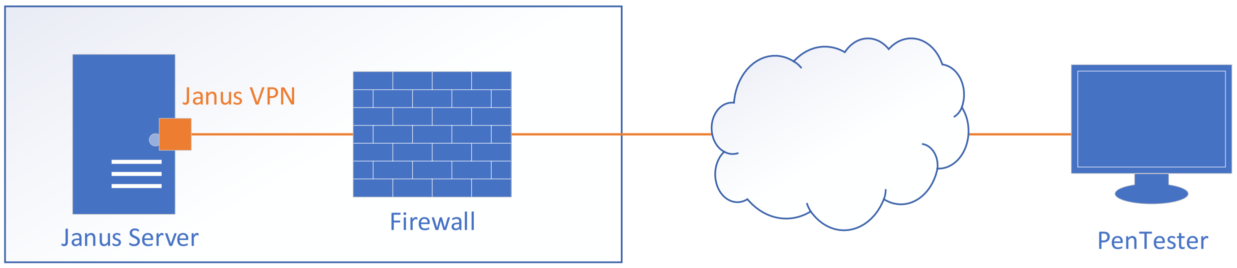
9.12. Compatibility with existing standards

The VPN product as described in the previous section operates as TCP/IP level and does not intervene with protocol levels above TCP. Thus, any communication protocol using TCP/IP can be forwarded by the VPN product.

It is the intention to construct products operating at higher protocol levels. In that case, compatibility has to be considered.

9.13. External penetration test

To facilitate a penetration test on our VPN product, we installed it on a virtual machine and configured our firewall to forward an external TCP port to the instance of the VPN product. This mimics a potential real-life operations setup.



Two independent parties were asked to perform a penetration test. Both were frustrated as their tools were not even able to determine what application they were testing and not able to find any issues what so ever, which is very unusual. Also, their testing did not disrupt the operation of the VPN product, its functionality was verified by using VPN tunnels when the penetration test was ongoing. The VPN product was fully responsive and operational throughout both tests. This is obviously good news.

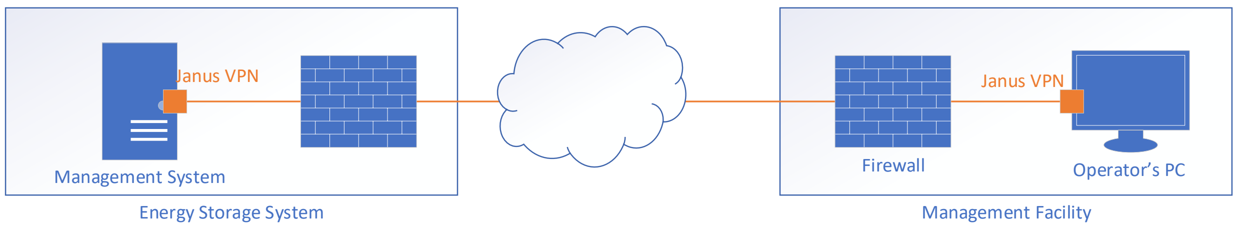
It should be kept in mind that a “clean” penetration test is not a guarantee of a perfect product, in particular in this case the techniques and tools used were of generic nature and not tailored our product and protocol.

9.14. Demonstration

The Janus VPN was demonstrated at Nerve Smart Systems for establishing a secure connection between the management facility and the management system at an energy storage system.

The benefit of using the Janus VPN firewall on such setup is to reduce the attack surface – in particular of the energy storage system.

The demonstration setup is illustrated below.



The demonstration was successful and the system worked fully as intended.

9.15. Technical results

The software of the VPN product described in section 9.11, which also was used for the demonstration, only had 1,298 effective lines of code. This is far below the target of 25,000 lines.

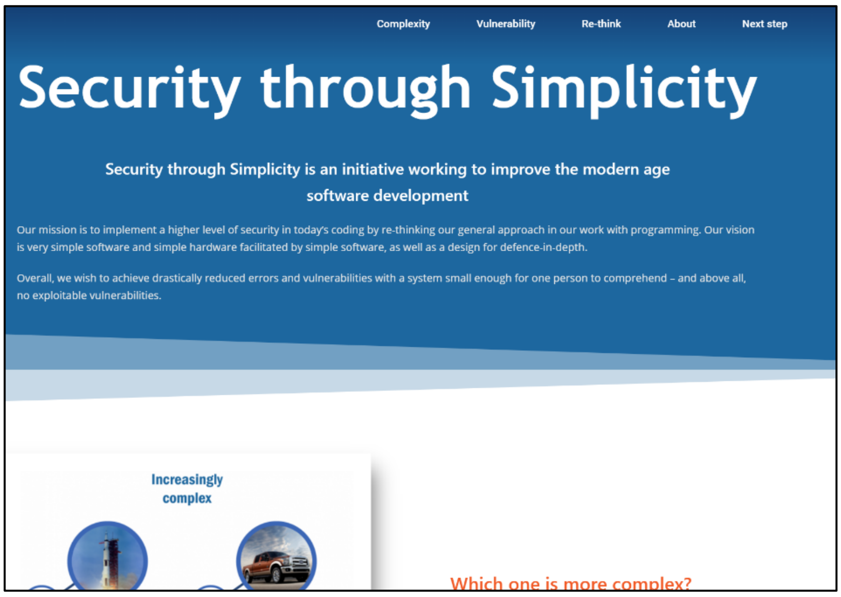
Effective lines of code mean only counting lines which can change the software's state. Empty lines, comments, function declarations, lines only containing '{' and/or '}' are not included. The code base has separate code for Linux vs. Windows in some places. In the counting, the code for both platforms have been included even though only one part is active at a particular build.

The line count above only include the code actively compiled into the demonstration software. Thus, other variants, various test code (e.g. unit tests) and build code is not included either.

9.16. Dissemination

The dissemination strategy of the Janus project was realised through a website, participation at conferences and numerous talks with industry actors.

The website was setup early in the project in order to establish a focal point for contacts for interested external parties. The site has been maintained and refreshed during the project.

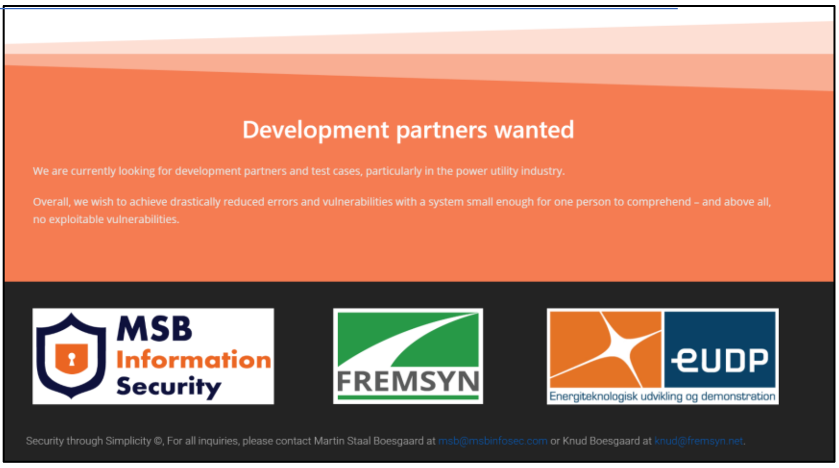


Security through Simplicity

Security through Simplicity is an initiative working to improve the modern age software development

Our mission is to implement a higher level of security in today's coding by re-thinking our general approach in our work with programming. Our vision is very simple software and simple hardware facilitated by simple software, as well as a design for defence-in-depth.

Overall, we wish to achieve drastically reduced errors and vulnerabilities with a system small enough for one person to comprehend – and above all, no exploitable vulnerabilities.



Development partners wanted

We are currently looking for development partners and test cases, particularly in the power utility industry.

Overall, we wish to achieve drastically reduced errors and vulnerabilities with a system small enough for one person to comprehend – and above all, no exploitable vulnerabilities.

Security through Simplicity ©, For all inquiries, please contact Martin Staal Boesgaard at msb@msbinfosec.com or Knud Boesgaard at knud@fremsyn.net.

Fig. 9.1 Screenshots (top and bottom) from the project website. [www.securitythroughsim-plicity.com](www.securitythroughsimplicity.com)

In January 2019 the Janus Project was invited to exhibit at the Homeland Security conference in Ingeniørhuset in Kalvebod Brygge in Copenhagen. At the conference Martin also presented an overview of the findings of the project in the proceedings.

Security through Simplicity
www.securitythroughsimplicity.com
Increasingly complex
Increasingly vulnerable
1969
145.000 Lines of Code Saturn V rocket
1981
400.000 Lines of Code Space Shuttle
2006
20.000.000 Lines of Code F-35
2018
150.000.000 Lines of Code Ford F150
RE-THIN
SOFTWA
DEVELOPMEN
REDU
COMPLEXIT
PII Guard

Fig. 9.2 Martin next to a Security-Through-Simplicity poster at the exhibition at the Homeland Security conference at Ingeniørhuset in Kalvebod Brygge in January 2019.

9.17. Commercial results

The Janus project has successfully demonstrated the concept of security through simplicity, and this has been discussed with several actors in the industry. Mainly in Denmark, but also internationally with contacts in primarily USA.

The project has not resulted in any contracts yet, but several leads are under way as described in section 10.1.

9.18. Future work
To further improve security, the system should operate at a higher protocol level than TCP. However, to achieve that, it must be compatible with existing protocols. One solution for this could be to split the security application into two parts – a "JANUS" part facing the internet, where this version communicated using simplified protocols, and a "conventional" part which has existing protocols implemented. This approach allows to interface existing systems even though that involves high-complexity protocols and at the same time allows for a simplistic interface towards the internet, where the biggest threat comes from.

We have started on constructing a framework for automated code generation for persistation and communication protocols. The intention is to allow for designing objects for persistation and communication in a web-based GUI and then, based on the configuration, automatically generate the code implementing these. One potential challenge here is to ensure the quality of the generated code – as part of this, the code-generating code should itself be small and easy-to-understand. This work should be finished.

10. Utilization of project results

10.1. Future commercial steps

There has generally been a good response and a high willingness to discuss the rationale and concept behind the Janus project among professional actors in the Danish cryptography sector – and also with several interested partners for purchasing the system.

The discussions are currently on hold due to the corona crisis, but they will be continued after the crisis.

- There have been several discussions with the partner where we ended up testing the system. They produce decentral BESS systems and are interested in using the system to secure control communication to the decentralised units.
  The experiences from the testing cycle is part of the foundation for the EUDP supported ongoing TOPChargE project.

- A large Danish manufacturer of electric metering devices is interested in utilising the Janus project to secure control and data communication with the external devices. The discussions are on hold currently due to the corona crisis but will be continued later.

- The product could potentially be of interest to the national Danish Center for Cybersikkerhed.

11. Project conclusion and perspective
The JANUS project has successfully developed and demonstrated a novel security product similar to a VPN tunnel for protecting Industrial Control Systems (ICS), e.g. SCADA systems, against cyber-attacks.

All objectives are fulfilled:
- The 'security by simplicity' is implemented in only 1.298 effective code lines.
- A 'defense in depth' concept with several security layers is implemented
- Two external penetration tests did were unable to compromise the system.
- The VPN product operate at TCP/IP level and does not intervene with protocol levels above TCP. Any communication protocol using TCP/IP can be forwarded by the VPN product.
- The product was successfully demonstrated at the Nerve Smart System BESS product.

The JANUS project thus successfully achieved all objectives, and the concept is ready to move on to commercialization. During the project period it was demonstrated that the product is very fitting to smaller distributed smart grid applications like smart meters.