1 Summary

This paper is loosely based on a lecture with the same title given by Antti Vähä-Sipilä as part of the course “Software Security” with Henri Lindberg presenting a sort of *from the trenches view* to the challenges typically seen while trying to uphold security mindset in a software project.

In addition to the horror stories about how the project can be completely mismanaged to ignore any security what so ever, the rest of the lecture presented various activities which are typical parts of a software life cycle and how security might be improved during these activities using suitable tools and skills.

2 Stories from the Trenches

As an introduction to how security might be introduced to a software life cycle Henri Lindberg talked about the contradictory view, how the security will never become a proper part of a software based product, how the security can be and all too often also is undermined through lack of vision, mismanagement, ill-advised attempts to saving money such that one ends up saving pennies in the short term while later on burning thousands and thousands of euros/pounds/dollars to plug holes which should never have emerged at all.

Especially for those attendees in the class who have not been working actively in the business yet, and who do not have first hand experience in software projects or in large scale integration projects this part of the lecture was supposedly good preparation to what might be coming in their way in future.

For someone with more experience in the current IT business this part of the lecture did not really give a whole lot of new insight, though, it confirmed once again what was already known from one's own experience.

The challenges in defending the security point of view tend to be heavily financial and political. Putting effort in security is seen just as a cost factor and as slowing down the pace of new software releases. People who are not part of the IT security scene themselves typically do not see security as an intricate part of any product or system but instead as some sort of wallpaper that gets laid on top of the product when the core functionality has been put in place.

Though this course concentrates on the security aspects this confused attitude is largely the same towards any non-functional requirements and features.

Independent of whether we are talking about performance, security, privacy, scaleability, service continuity, etc. people working in other branches of life are usually relatively eager to concentrate on the functional features and requirements, but they simply refuse to bother when they should discuss qualitative features.
What is considered good performance?
How should the system be scaled when necessary? - Scale up or scale out?
How much can the scaling cost?
What sort of threats must the system withstand?
How determined attackers shall the system withstand?
What is the business impact if confidential data gets leaked?
What is the need for service continuity?
Shall an integrated environment continue working if it gets partitioned in the middle?
If partition tolerance is supported, shall we prefer consistency or availability?
How shall various parts of a system react when given unexpected input?

In everyday life people have an enormous tendency to ignore the hard issues and concentrate on the easy ones to make their mark on something.
Lately the management in various enterprise environments has just plain abused the expression *agile development* as an excuse to do sloppy work and avoid committing to specifications and designs. This obviously drives any projects in totally ad-hoc mode.

This is the mindset which defines project timelines, project budgets, and the requirements specifications. When there is no time, no budget, nor proper requirements for non-functional or qualitative features, their design becomes ad-hoc as best while the functional features are typically much easier to design. Where the design is sloppy, the implementation by necessity becomes sloppier. Testing a feature which has not been specified tends to be left undone completely. How could anyone create a test case for something which has not been specified?

Looping phases through implement, test, and fix does not always work even when there is a proper specification and a proper design, but lacking specification and design, repeatedly trying over again is not likely to improve the quality of the outcome.

When the information security budgets are often flat 0, the requirements are securitywise low grade, and nobody has the security issues as primary goals on ones agenda, everyone tends to be completely overwhelmed with other unrelated tasks. With nobody to drive it security becomes easily ignored and forgotten. In this sort of environment, instead of continuous security assessment through the whole life cycle, the security work gets quickly reduced to an imitation of vulnerability assessment far too late right before shipping the product or to none at all.

This sort of environment was described by the speaker with a quote from TV series character Dr. House: “Treating the illness, not necessarily treating the patient.”
The illness in this case are the low grade qualitative features of the software. The patient is the process which creates sloppy products.
If we want to improve the products, we really need to start fixing the process.
3 Changes to Mindsets to Alleviate Problems

There are a few things we can try to make things better…

1) Make buyers understand that a security assessment may be much more expensive than the actual software.

2) “Agile project” means exactly the same as iterative stepwise refinement using small manageable size steps. It is not a good reasoning for Wild West mentality.

The author of this paper once got a relatively big boss at a very big client very, very angry indeed by pointing out this corrected interpretation of being agile.

To avoid further trouble, keep in mind that caution and soft diplomatic touch may be required.

3) Make sure vendors follow processes and best practices instead of using agile as a cover for ad-hoc work.

4) Make project governance boards understand yet another quote from Dr. House: “Treatments don’t always work. Symptoms never lie.”

Applying a fix is not enough, if one cannot prove the symptoms have disappeared.

Proving the existence of a vulnerability is often the hardest part. People do not wish to see the problem, because it destroys their feeling of achievement. So, they are prone to play down what ever symptoms they are shown calling them a lucky shot, an isolated case, or very rare, etc.

One may need a proper automated exploit which succeeds almost always to convince people that this or that particular nit needs to be fixed. This is very labour intensive for the security personnel.

4 On Certificates and Certification

Lately all sorts of certificates have become very popular means to indicating either personal skills or companies being able to follow a certain audited process. Especially on the individual persons' context this is probably also a symptom of the fact that various security curricula tend to cover very distinct content, and potential employers and clients do not really understand what to expect a person to master when (s)he has graduated from a certain institution. The logic apparently is such that the content of a certification test is easier to understand in a CV.

For enterprises the certificates are similarly part of the marketing material they use to convince potential clients about the quality of their products and services.

It is worth remembering, though, that a certificate may not mean a whole lot. First of all a certificate may actually indicate knowledge of another area entirely. Secondly all certificates are not created equal. Not all of them guarantee true expertise, true quality, etc.

All too many certificates only guarantee that the company running the certification programme is in the business of making profits for its owners.

Proper attitude and leadership are the most likely things to making any lasting impact.


4.1 A Little Anecdote about Certification

The author of this paper once worked for a company named SysOpen which had an ISO 9001 quality certificate. Having that and being able to maintain it year after year was quite a benefit for marketing. Supposedly the clients never quite figured out what was actually being audited. That ISO 9001 never covered anything else but 3 project documents: a project start document, a project end document, and an optional anomaly/exception report which was filled on monthly basis had there been an exception from what was agreed with the client before the beginning of the project. Obviously the exception document was not used very often.

This was an absolutely brilliant marketing tool!

This should also serve as a good reminder that the any standard compliancy certificates tell only so much. Most of the whole standard may be rendered as N/A (= not applicable), except where applicable auditing compliancy may be very straight forward.

Similarly a standardised audit tied tightly to Microsoft models of doing things may actually lead to a situation in which most of the standard reference model does not apply to UNIX vendors at all. In such a case becoming audited may be relatively painless, but the certificate does not guarantee much about the UNIX vendors practices and processes.

5 The Software Life Cycle and Security

After the visitor from industry had given his speech, the actual lecture part began with a sample catalog of activities in a software life cycle which were then later connected to various security oriented tasks.

As models of far larger catalogs of activities which often take place in a software life cycle two alternatives were mentioned BSIMM (Gary McGraw, 110+ distinct activities) and its open relative SAMM (Software Assurance Maturity Model, OWASP).

Two proper standards guiding security related tasks in a software life cycle as examples of official standards for which audited compliance might be required in the future: ISO-27034-1 (very much geared towards Microsoft architecture) and Finnish VAHTI 1/2013 (application security, which of course is moot unless the platform is equally solid).

The list of software life cycle activities presented during the lecture was:

- requirement specification (product management),
- design,
- implementation (coding),
- QA/testing,
- project management,
- documentation,
- shipping/deployment,
- user training,
- operations,
In today's world most of these activities are no longer just a single temporal instance but instead often ongoing activities throughout the life of the product. Essentially it is often hard to speak about a software project or sub-projects within the life cycle. Constantly on-going activities are processes, not projects. Just as a model of thinking one could also compare these activities to the deep parallel pipelines in super-scalar CPUs. Various phases of instructions (fetching instruction, decoding instruction, fetching arguments, execution, fork prediction, storing results, etc.) keep happening at the same time as parallel activities and then passing the intermediate results to the next stage in the pipeline until an instruction has been completed. When an instruction eventually completes in the last stage of the pipeline, at the same time there are already parts of the next instruction being processed in the earlier stages of the pipeline.

5.1 Co-operative Development vs. Strict Divisions

Some people like to talk about continuous delivery or DevOps meaning at least partially the same thing. The important things to notice in the picture are anyhow the fact that since a lot of the activities are practically constantly on-going there are no savings in trying to keep the people driving these activities somehow compartmentalized under distinct umbrella organization or units. Instead sticking to too much isolation between the personnel driving this or that aspect will only reduce communication and reduce quality by assuming security is someone else's job. In a security-wise successfull environment everybody should be doing security work independent of one's primary task or activity.

Notice, by the way, how detrimental is a manager who concentrates only on one's own organization unit, accounting, and cheese cutter method to gain savings! The author of this paper has been unlucky enough to have seen a number of these life forms.

On the other hand the most psychologically rewarding "projects" witnessed by yours truly have been ones in which the project teams may have been representing various very autonomous units (development, operations, security, finance, …) in very large organizations with also some externals and even users in the team. When such a joint team has been able to work freely as a self organizing shadow organization unit of its own without the various department heads trying to get their way in everything, the eventual resulting integrated setup has been usually performing much better, scaling better, and even any third party security assessments have resulted in far less negative feedback for security issues.
6 Activities, Security Related Tasks, and Tools

All in all total software security is impossible or at least impossible to prove conclusively. One may get to an epsilon distance of it but assuming having reached it is fallacy. When time passes, the environment gradually changes, and the epsilon grows.

Let us assume here that the production at hand is so big that the initial release cannot be built just scrumming iteratively. Much of the activity can still proceed parallellized in pipeline manner, but there is just too much personnel involved to work as a single small scrum or DevOps team.

6.1 Requirements

Requirement specification does not really benefit of using any certain tool but having a kind of a target model in mind will certainly help.

The requirements should be clearly divided into functional and non-functional (qualitative) requirements. The latter set should contain statements such as

1. requirements for authentication,
2. requirements for encryption (what will be encrypted, strength, PFS = perfect forward secrecy, ...),
3. requirements for signatures,
4. need to comply with standards or specifications,
5. other required tests/audits (abuse cases, attacker stories),
6. security requirements for the platform (incl. CAP = consistency, availability, and partition tolerance), ...

6.2 Design

Engage a security specialist!

Detailed threat modeling / architectural risk analysis must be part of the design. Unless and until this task is done, a project should not proceed to implementation.

Test cases at least for any eventual system level tests should be detailed by the design crew or by a separate ”tiger team” working out of the requirements and design documents.

One way of keeping designers who are not part of the development crew active after the initial design has been passed to the developers, is transforming them to a tiger team working on an automated test rig which will be used to run all system level tests (often called acceptance tests).

Those test which cannot be automated should go in a checklist much like those used by airline pilot before takeoff and landing.

Some people from the design crew could double as a documentation team.

Design quality benefits most of being reviewed by someone who was not part of the design crew but in theory could have qualified as a member of the design crew. This is especially true for the security features.
6.3 Implementation and Testing

Like everything else also security features are born in this activity.

Yours truly has never accepted testing or QA as a separate phase of software life span. Implementation and testing are – and must be – inextricably intertwined.

The testing activity is a combination of module testing, integration testing, and system testing. Module testing of the higher level modules naturally serves also as integration testing for the lower level modules.

Everybody implementing anything should be taught to module test every little code snippet before putting it aside and tackling another.

System level testing is nothing more than the integration testing of the full integrated environment.

Sometimes it helps, if module tests are written by a fellow programmer. We are all prone to be blind to our own blunders.

Especially it takes a very disciplined programmer to consistently include also faulty input to ones test cases.

For any tests there should always be a separate test driver and all test cases should be added to it such that the same test cases get re-run every single time something is changed. The same test must also be repeated quite independent whether the change was a bug fix, a new feature, or an optimization.

For module and integration testing there is a wealth of tools both static and runtime/dynamic to find potential sources of security issues and other errors.

Static analyzers are in a manner providing automated code review either by reading the actual code or by reverse engineering the binary. Anyhow, they are definitely not just the "glorified grep" as one snarky remark during the lecture put it.

Static analysis tools provide e.g. code coverage analysis, data flow analysis, code complexity analysis, limited memory analysis, etc.

More complete memory management monitoring and realistic test coverage monitoring, to see which functions have been activated by the use of a certain test input and which have not, needs runtime instrumentation.

The problem with many of the commercial tools, though, is the fact that they are usually available only to a limited number of platforms and often they tend to be costly.

Yours truly remembers seeing quite nice output from e.g. Purify and Veracode, both of which have traditionally put a heavy burden on the old wallet and which have not been available for OSX and FreeBSD.

The best static analyzer which the author of this paper has ever seen proposed refactoring when code was too big and/or contained too many nested control structures. It even gave very accurate estimates of the time used to write the analyzed C files. Being a commercial product it also was available to only a few systems (including Solaris). Because neither OSX nor FreeBSD was among the supported target environments and because several years have passed since the encounter, the name of the product as already been lost in the memory lane.

Being a long time C affectionado, yours truly has had a long and fruitful relationship with e.g. these free tools:

- ElectricFence, dmalloc – malloc() dynamic memory management
- rats, flawfinder – look for potentially dangerous function calls
truss, ktrace – system call traces

tcpdump, wireshark – protocol traces

lsof – open file descriptor diagnostics

FreeBSD ld and C library – alerts when a program uses unsafe functions

Relatively recently there has been a very influential new kid on the block, though: clang 3.4 provides both static analysis (--analyze, -Wall, -fsyntax-only) and runtime instrumentation (-fsanitize*).

Clang beats both gcc and any lint flat out in the verbosity and accuracy of any errors and warnings. Clang also alerts for things which gcc and lint do not even notice. The level of notifications given by Clang is not maybe exactly equal to Ada or Eiffel but at least getting close.

Anyhow the important thing is to use rather too much tools and instrumentation than none at all and being humble when the tools alert.

6.3.1 The Human Factors

In addition to using a bunch of technical tools for quality assurance and really paying attention when they notify of something there are also relatively simple and sane rules of thumb which can make a big difference for the developers.

The first one is the very common wisdom that an average human brain can only handle some 5 (4-6) items at a time. Sometimes this gets also expressed in the form of average humans being able to handle some 5 levels of indirection at best. If one makes the mental leap to see that any condition or control structure in the source code hogs one of these levels and each action being controlled by those structures eats up yet another level, it should be obvious that an average human can handle relatively reliably only some 3 nested control structures (if, for, while, switch, case, etc.). When deeper nesting is needed refactoring to separate functions is certainly beneficial for both development and maintenance.

So, you are one of these exceptional persons who can handle some 8+ levels of indirection at once. Good for you! Still, if you use all that capacity, the percentage of the humanity who can work with your code got just reduced to about 2% instead of 50% making your code practically unmaintainable.

The second easy rule of thumb is the size of modules in lines of code. In 1990 Carol Withrow published a finding [Wit90] that on the average humans produce best code quality when the modules are around 200-250 lines in size. Larger modules become too big and complex to keep all the details in mind. Much smaller modules have two handicaps. Plain human arrogance may cause small modules be thought of as somehow obvious even when they are not. On the other hand mistakes with call interfaces (APIs) tend to be distributed evenly over all function invocations. The effect gets highlighted in small functions calling multiple other functions.

Though, Withman's original reference material had been written in Ada, as long as we are discussing human mental abilities, the programming language should be largely irrelevant. Certain compilers or interpreters can only help by raising the developers' alertness in face of potential blunders.

There are also other human issues like the importance of style, affecting software quality. It is worthwhile to read "The Practice of Programming" by Kernighan and Pike [Ker99].
6.3.2 System Testing

Once there is a full integrated system to test and the design / tiger team specified tests have been successfully completed, burning the system properly using various fuzzers will certainly be worth the time and effort. Use at least radamsa, the one tool which was presented during the first lecture on this course and with which everybody should now be familiar.

How one gets the message through to a finance officer on a savings mission, seems to have a tendency to become a bit of challenge, though.

6.4 Shipping / Deployment

Putting the product to some real use should be relatively straight forward once both the software package and the target platform have been cleared through their acceptance and compliance tests.

There are a few tricks, though, which based on the author's experience are very beneficial for all security and maintenance of the application as well as for system administration.

One should clearly understand that in any installation there are potentially five different types of objects to handle:

1. the program binaries themselves,
2. application data (e.g. static web pages, templates, preloaded databases, …),
3. configuration items needed by the binaries,
4. cryptographic and signing keys, and
5. any licences required by 3rd party applications.

These distinct items have to be packaged separately, but also dependencies between versions have to be handled. It does not take a huge mental leap to figure out that mismatched binaries and configuration may cause a royal leak in the system.

On the other hand one does not wish to change the configuration when the binaries get updated, unless it is really necessary. An unnecessary change is always a potential source of instability. The result could be nothing at all, only reduced availability of the system, or an unexpected backdoor in the system, or anything in between.

Similarly keys, licences, and service data may have to be updated without risking derailing the rest of the installation.

6.5 Operations, System Administration

This activity tends to become a boiling hot-pot. Since the platform like OS and any potential middleware needs to be regularly updated to avoid them becoming some sort of gaping hole allowing bypassing any designed protections, there is always a risk of the platform and the application eventually becoming a mismatched setup. Have at least one test environment in which there can always be the latest and greatest OS version.

Assuming the application does not require huge amounts of hardware resources, on top of that test hardware one can run a bunch of light-weight containers or jails in which there could be distinct middleware and application versions for testing.
until both developers and administrators are convinced of the solidity of a new setup.

In the best case there could be a spare hardware environment which could be updated to become the new production environment and tested while the old one still keeps on providing the service. When the new production setup is ready one either could disconnect the old production environment from the network and assign its production interface address(es) to the new environment. Alternatively when the whole setup needs to live behind a firewall anyhow, one could simply change a NAT (= Network Address Translation) rule to point to the new environment instead of the old one. The users of the service would hardly notice the interruption of the service.

Obviously the previous production system should now become the spare and be updated to the latest and greatest versions of everything.

6.6 Maintenance

In maintenance mode almost any application can be developed in true small agile scrum or DevOps team in which the roles are complementing one another. Still the activities are basically exactly the same as in a vanilla software project.

If there are new requirements which need more than a skeleton crew to work, it is still usually going to be a relatively small team. The product manager and the requirements team filter the ideas coming from the users/clients and operators. When the amended requirements are in place. The design will most likely become a team effort within the skeleton crew.

The implementation and new system test cases should preferably be kept with separate team members, the changes needed in the production platform might need to be assigned to a third person. Otherwise there is nothing much that needs to be fixed to any single individual.

6.7 Documentation, User Training, Customer Management

All the training material and documentation has to state clearly any potential security risks the clients / users could cause through their actions and be warned and advised against such actions. On the other hand any good system should have been designed and built against faulty user input.

If anyone can cause a real disaster, the opportunity lies with the operators and system administrators. These life forms most likely have access rights which can completely bypass the applications and fiddle directly with their data. Still it may be necessary to point out we are now discussing remote system administrators at the clients’ own sites and assuming that the application has been installed in the clients’ systems.

In-house administrators who have a first level contact with the development team are ideally part of the development team. Thus they probably fumble less and talk to the team more.

Especially small companies tend often outsource their system administration to a varying set of people who anyhow undertake potentially competing or only partly understood maintenance tasks. Such system administrators are really a disaster just waiting to happen. For these guys a quick introduction to care and feeding of a ”new family member”
is absolutely necessary.

6.8 Sales & Marketing

This is pretty much the same problem as with documentation and training with the distinction that the documents and training is potentially a talking head and the authoritative reference just hearsay.

This package is poisonous because the sales and marketing people often say what they think a potential customer wants to hear instead of finding out the real facts.

We should probably require that all marketing material has to pass through a security review and all sales people must pass an internal security training for any application they are expected to promote, but knowing how bloated egos the marketing people often have, these requirements might be left as just that with no compliance unless the alternative is getting the proverbial pink slip.

If anything, this is someone else's problem, the someone else being the company's CEO who has to officially set the rules and uphold them.

7 Symptoms of the Lack of Necessary Security Skills

In many fora, including the SANS NewsBits news letter[14], there have been lately good examples about how the lack of necessary security skills manifest in environments which have understood their profound need for cyber security.

In some cases symptoms have been just complaints of lack of sufficient education and training.

Other examples have voiced concerns about “tools skills” being neglected in education in favor of more general scientific skills. Of course this category is basically the ages old discussion about the merits of vocational vs. academic education which is also completely ignoring the fact that when someone has broad enough general understanding of an issue, this or that new tool can always be familiarized with later on, if needs be.

A third line of symptoms seen in media is how to recognize talent, skills, and aptitude of people regarding the IT security issues and how to hire competent people for security oriented jobs.

Sad as it is, the big problem with the current discussion around security skills is that it still revolves completely around the factory floor and blue collar personnel. Nobody seems to be openly discussing the lack of security skills in the higher management of organizations. Even the best skills in the lower levels do not mean much, if the higher echelons undermine the work of the grass roots level.
8 References

1. Lecture 3 outline, Security in a Software Project; 
   https://wiki.helsinki.fi/download/attachments/127960693/Session3.pdf?version=1&modificationDate=1395841684183&api=v2

2. "Secure Coding, Principles & Practices";
   Mark G. Graf & Kenneth R. Van Wyk;
   O'Reilly & Associates Inc, June 2003

3. [SANS14] SANS NewsBites Vol. 16 Num. 024; 
   March 25, 2014; 
   SANS e-mail news letter

4. [Wit90] "Error Density and Size in Ada Software";
   Carol Withrow; IEEE Software, 1990

5. [Ker99] "The Practice of Programming";
   Brian W. Kernighan & Rob Pike;
   Addison-Wesley, 1999