Technical report
Source code audit of Norwegian electronic voting system
Ministry of Local Government and Regional Development

Tor E. Bjørstad, <tor@mnemonic.no>

Oslo, 2013-08-07

mnemonic
- securing your business

Executive summary

On September 9th 2013, parliamentary elections will be held in Norway. For the 2013 elections, 12 of the 428 local municipalities in Norway will offer their citizens the opportunity to vote in advance over the Internet, as part of a trial project organized by the Norwegian Ministry of Local Government and Regional Development.

Internet voting has previously been piloted in 2011, with ten municipalities participating in a trial run for the local elections. The system is based on an advanced cryptographic voting protocol that has been designed to maintain the security, anonymity and verifiability of votes cast online, while requiring a minimum of trust between system components.

The Norwegian Ministry of Local Government and Regional Development hired mnemonic to perform a "*third party review of those parts of the [electronic voting system] that implement cryptographic primitives and generate keys*", to verify that this is done securely and correctly.

mnemonic has not discovered any critical cryptographic weaknesses that would preclude the use of the Internet voting system in the forthcoming election.

The most serious technical issue discovered during the review is an error in an encryption format for storing password-protected data. If patching this bug before the elections is deemed too risky, we recommend compensating with operational safeguards, such as strictly limiting and monitoring access to systems where such data is stored.

A number of other flaws and potential security issues have been uncovered, some of which require further analysis by the vendor. Based on the audit as a whole, and our current understanding of the voting system implementation, mnemonic would like to suggest four main recommendations:

1. Address the security issue(s) caused by cryptographic weaknesses in the password-based encryption function `cipherSymmetrically` in the `UtilsRBAC` class (4.3.2.1).
2. Ensure that sufficient safeguards are in place to maintain the integrity of the audit events, despite potential vectors for dishonest parties to truncate or manipulate data from the secure log appenders (4.4.1).
3. Verify that observations made relating to key generation (4.5) do not threaten the overall security of the protocol, by inadvertently revealing keys, passwords or other security-critical information.
4. Take concrete actions to improve the overall state of the source code, e.g. by consolidating and refactoring duplicate code, cleaning up interfaces and dependencies, documenting technical architecture and usage, and continually improving quality assurance practices in the development process.

Key stakeholders should review the technical chapters 3 and 4 of the report in detail, and take appropriate actions to evaluate each of the findings presented, and assess and manage any associated risks.

Table of Contents

1 Introduction

On September 9th 2013, parliamentary elections will be held in Norway. In the weeks running up to the election, twelve municipalities offer their voters the opportunity for advance voting over the Internet, as part of a trial project organized by the Norwegian Ministry of Local Government and Regional Development.

The electronic voting project was originally started in 2008, and a previous trial was held as part of the local elections in 2011. General information about the project, including previous research and evaluation reports, is available at http://www.regjeringen.no/nb/dep/krd/prosjekter/e-valg-2011-prosjektet.html?id=597658.

The electronic voting system has been implemented by Scytl Secure Electronic Voting SA. The back-end system is written in Java, and the voting client runs as Javascript in the user's browser. To ensure transparency and foster trust in the solution, system documentation and source code is available to the public, at https://brukerveiledning.valg.no/ and https://sourcecode.valg.no/websvn/.

1.1 Project Mandate

mnemonic was hired by the Norwegian Ministry of Local Government and Regional Development to perform a "*third party review of those parts of the [electronic voting system] that implement cryptographic primitives and generate keys*", and verify that this is done securely and correctly. The review has been carried out in the period from July 9th to August 5th, 2013.

1.2 Scope and limitations

This review uses a bottom-up approach to analyse the cryptography used in the Java implementation of the electronic voting system. The scope of this review is thus quite narrow, and we wish to emphasise this fact strongly at this point.

As a consequence, there are several things that this report is *not*, including:

- a general code review covering functional aspects of the source code
- an audit of the election system configuration
- a risk assessment of the entire e-voting system
- a cryptographic evaluation of the secure voting protocol
- a political statement for or against online voting

There are also several interesting areas that this report does *not* cover, including:

- the Javascript voting client implementation
- operational aspects of how the voting system is used
- the security of technical infrastructure, including operating systems and networks
- trust relationships with third parties, such as the identity provider (Difi/ID-Porten)

In a technical source code audit it is easy to focus narrowly on negative findings, deviations and errors, and it is worth pointing out that the Internet voting system appears to be working well from a functional point of view, while implementing a very advanced, complex, and distributed cryptographic protocol. This is in itself a significant achievement.

The central mission of this review has been to carry out a technical review of those parts of the Java source code that implement basic cryptographic and security-related functionality. In such a review, the key focus is on the correctness and appropriateness of the low-level implementation. As a consequence of this scope, the high-level "bird's eye" perspective of how the different low-level primitives are composed and utilized by the protocol is not always present.

As a secondary output from the audit, the report also presents more general criticisms and suggestions for improving the overall quality of the e-voting project. These are both with respect to the cryptographic implementation, and for the overall software quality, and are to a greater extent subjective assessments, based on prior experience with the security of comparably large software development projects.

Primary audiences for this report include project management, cryptographers, security architects, developers, and other key stakeholders, both in the e-voting project and at Scytl. It is also expected to be of general public interest. However, readers should be aware that the main parts of the report (presented in Chapters 3 and 4) assume a high degree of technical familiarity with the Java programming language, cryptographic techniques in general, and the e-voting project and protocol in particular.

1.3 Structure of the report

The report is divided into five main chapters, and an appendix.

1. The current chapter provides context and introduction to the source code review, and is useful as a less technical summary of the work that has been done.
2. Chapter 2 presents an overview of the activities performed during the course of the project, along with a description of the methodology, tools and overall approach.
3. Chapter 3 provides an introduction to the e-voting source code, and contains general technical remarks and observations from the review.
4. Chapter 4 is the main technical chapter of the report, and provides analysis of core cryptographic packages in the e-voting project. The analysis is organized by Java package and class, starting with basic functionalities and moving towards more advanced applications.
5. Chapter 5 concludes the report, and summarizes the main takeaways and recommendations from Chapters 3 and 4.

A. The Appendix provides a list of all specific findings and observations.

1.4 Summary of findings

Findings in the report include more than 40 specific observations that may have a potential security impact, as well as the (positive) validation of several of the implemented primitives. These are described in Sections 3 and 4 of the report, and summarised in the Appendix.

It appears that our most serious finding is related to a password-based symmetric encryption scheme defined in `UtilsRBAC.java`, which contains multiple cryptographic weaknesses and can leak information about the encrypted data. The class is used in several places, including during key generation. It should be noted that this encryption format appears to have been defined outside the scope of the Internet voting implementation project.

Several observations are made regarding the jbasis library, which is a shared library written by Scytl and used across many of their projects. The library contains several default settings that are insecure or deprecated, and should not be used in new applications. Much of this is likely to be due to backwards compatibility, but it means that parts of the library must be used rather carefully to maintain security.

The secure-logger library used for auditing could in some situations permit an attacker with file access to truncate and modify the logs, without this being clearly detectable within the cryptographic framework. This is probably easiest to prevent through operational means, such as a system for log monitoring and collection located in a different security zone. The Norwegian system uses Splunk for this purpose.

Regarding the key generation routines used to set up an election, we have noted specific areas where one must be careful to prevent sensitive information from leaking, for instance due to master passwords being written temporarily to disk. In practice this should take place on an isolated system with very restricted access and an encrypted hard drive, though these are once again operational, rather than cryptographic, safeguards.

On a more general basis, it appears that the overall quality level of the Internet voting system implementation has room for improvement, and there is a legitimate concern that the overall complexity level of the system, including complicated and tangled dependencies between different parts of the code base, may serve both to cause and to conceal security flaws.

1.5 About the author

Tor E. Bjørstad holds a Ph.d. in cryptography from the University of Bergen, and has professional experience from some of the largest software development projects in the Norwegian public sector in recent years. He has worked full-time as a cryptographer and security expert since 2006, and has extensive experience with analysis of cryptographic primitives and protocols, security standards, and application security testing.

2 Project Description

Conducted by	mnemonic as
Performing consultant	Tor E. Bjørstad
Point of Contact	Christian Bull, Ministry of Local Government and Regional Development
Project started	2013-07-09
Report presented	2013-08-07

2.1 Methodology and tools

The source code review has not followed a strict formal methodology, though the overall approach is inspired by the OWASP Code Review Guide[1]. Work has consisted of two main phases; an initial period reviewing documentation and mapping the code base, followed by the source code audit proper.

Initially the full Internet voting source code was downloaded and built, on a local Linux VM. The first week of the review was spent getting acquainted with the project documentation and the overall structure of the code. During this period, static analysis tools were applied to identify potentially troublesome areas. Time was also spent to manually identify modules implementing core cryptographic functionalities, as this was not explicitly described in the project description or documentation.

The latter phase of the project consisted of the source code review itself, in which the code has been reviewed package by package, and the findings (whether positive or negative) have been documented in this report.

2.2 Project execution

During the initial phases of the assignment, it quickly became clear that the scope of the review would be far greater than the estimated "*3-5000 lines*" of source code that was indicated in advance of the project start. In fact, looking at two of the core cryptographic packages, `com.scytl.evote.protocol.ciphers` and `.signers`, they comprise *by themselves* 3000 lines of actual Java code (after discounting the licence header, blank lines, and comments), and more than 5500 lines in total. Furthermore, this is only a small portion of the code that has been covered by this review. As a consequence, the evaluation has taken somewhat more time than initially estimated. It has also not been possible to have closer look at how the different primitives are composed in the high-level protocols, such as authentication, given the time available.

During the review, representatives from Scytl have been available for questions and clarifications. In particular, Sandra Guasch has served as a main technical contact, and she

[1] https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents

has given consistently rapid and helpful responses to technical questions, both to clarify ambiguities and judge the impact of potential findings. The overall impression of Scytl has been very positive in this respect.

The project has been particularly challenging to carry out, both because of a tight time schedule, high workload, and unusually high complexity of the systems under review. However, the report is still able to cover a lot of ground, and should serve as valuable input for those working to improve the system.

2.3 Areas not covered

Cryptography is pervasive in the Internet voting source code – to such an extent that it would be utterly infeasible to give every application of cryptographic techniques a full review in the time allotted. Because of this, the central focus for this review has been the low-level implementation and "core" functionality, rather than trying to find and check every place a crypto interface is used by application logic.

As discussed with Christian Bull at the initial project meeting, the new Javascript client implementation is outside the scope for this review. Our understanding is that this is because the client is not considered to be trusted – voters should be able to detect a cheating or malfunctioning client by verifying that return codes are correctly received. In parallel with this project, a separate security test of the web application has also been carried out.

The Internet voting source repository contains significant amounts of code that is not used in the production system – proof-of-concept code, legacy code that is no longer used, code used for unit and integration testing, free-standing command line verification tools, and so forth. Where we have been able to verify that the code is not part of the production system, we have not looked at it in detail. It is, however, important to ensure that e.g. insecure test code is not used in production by mistake or misconfiguration.

The security of third-party libraries, such as BouncyCastle, has not been reviewed. However, looking only at the dependency, it is worth noting that the BouncyCastle packages used are versions 1.44 / 1.45, whereas the latest release is 1.49. Similarly, spring-security-core 3.0.0 is used, while the newest stable versions are 3.0.8 and 3.1.14. There is a known security flaw in versions of spring-security-core prior to 3.0.5[2], though it is unclear if this flaw can be applied to attack the Internet voting system.

Third-party dependencies that are not security libraries have not been examined at all. As a rule, our clear recommendation is to keep third-party libraries patched to their latest stable versions, whenever this is feasible.

[2] CVE-2011-2894, see e.g. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2894

3 Source code review of Norway's electronic voting system

3.1 Structure of the review

The analysis consists of two main parts: a short description of the e-voting source code and general impressions, and a review of low-level cryptographic functionality and its basic usage implementing core election processes.

Java source and class names are formatted using a `fixed-width` font, and classes are referred to by package name, rather than by file path. Class names are generally (but not always) unique across packages.

3.2 Introduction to the e-voting codebase

The full source code for Internet voting is available to the public, and can be downloaded from https://sourcecode.valg.no/websvn/. It consists of 9 main parts: a set of tools for build and deployment, two supporting libraries, and six main projects. In total, the code base contains roughly 350 Java packages and 2200 classes.

Table 1 below indicates the different projects, and gives an estimate of their relative sizes, measured in the number of lines of code (LoC). The estimate does not count configuration files, unit tests, or third-party libraries. It also omits the size of comments and blank lines.

Project	Version	Purpose	Approx. size
parent-config	2.3.1	Scytl build configuration, no code	-
jbasis-parent	2.8.9	Supporting library implementing ”basic java” functionality, including cryptographic interfaces	24 000 LoC
secure-logger	2.0.6	Supporting library for secure logging	4 500 LoC
auditing	3.2.4	Auditing module	5 750 LoC
authentication	3.2.4	Authentication module	11 250 LoC
counting	3.2.4	Ballot counting	38 000 LoC
evoting	3.2.4	e-voting application	25 250 LoC
protocol	3.2.5	e-voting cryptographic protocol algorithms	34 500 LoC
vsframework	3.2.4	Voting system framework	68 250 LoC
TOTAL			211 500 LoC

Table 1. Overview of e-voting codebase

As we can see, this is a rather large project. It should be noted that the above numbers do include “dead” and unused code, proof-of-concept code, and various free-standing utilities that are not part of production code as such. This is because these still remain as part of the code base, and often not readily identifiable. One may also argue that the line count is a fairly rough size metric, though it does give some indication of the overall complexity.

Of the different modules, much of the cryptographic code is concentrated in the `jbasis-parent` and `protocol` projects, and this is where most of the report's attention will be focused.

3.3 General impressions

Internet voting is a large and complex topic, and as such the implementation has much in common with other large and complex enterprise software implementations. What makes the e-voting project unique is the pervasiveness of cryptographic techniques. Large-scale software engineering is challenging in itself, and it is unsurprising that large-scale cryptographic engineering is harder still.

In fact, one of the most striking observations from this review is to which extent the source code looks a lot like an *average* large enterprise software implementation, in terms of code quality, documentation, fragility, consistency and so forth. Perhaps this is a natural consequence of the scale of the project, yet perhaps also a sign that the focus on software quality might be expected to have been stronger, in a project touches the core of our democratic society.

As projects grow large, a certain amount of discipline and management is needed to keep the complexity under control and the project on track. This is self-evident for measurable quantities such as time and cost, but perhaps less obvious with respect to an intangible concept like "software quality". Yet establishing a structured approach to quality engineering and continuous improvement throughout the development lifecycle can yield clear benefits, particularly if started at an early stage and with strong management support.

From reviewing the e-voting source code, one gets the impression that many common practices of reliable and robust software engineering may have been lacking at some point in the development process, leading to significant technical debt. This may have been a question of time, of cost, or of priorities, and there may also be historical reasons that explain it. In the experience of the author, it is often also a question of culture.

General-purpose quality management frameworks (such as CMMI, ISO 9001, and TQM) tend to focus on high-level processes and documentation, rather than prescribing specific techniques or measuring compliance with those processes. Industry standards can be somewhat more specific; the PCI standards for applications handling credit card data specifically mandate formal code reviews, documented secure coding guidelines, and the removal of test data and passwords prior to release.

Well-proven techniques such as test-driven development, mandatory code reviews, static analysis, refactoring, use of checklists, and standards for documentation and code can all yield tangible quality benefits, not least with respect to non-functional aspects such as reliability, verifiability, readability, maintainability, extensibility, and other "-ilities". Some of these techniques may already be in use by the project, others may be less relevant, either way one should consider how current processes can be improved.

At the end of the day, quality engineering is a means to an end, and the goal should be to establish a project culture for delivering well-engineered high-quality software, rather than

adding an extra layer of bureaucracy by implementing some particular quality assurance framework for its own sake.

Within the field of security, there is a common saying that there exist only two kinds of software systems: those simple enough to be "*obviously secure*", and those complex enough to be "*not obviously insecure*". Secure electronic voting may be so inherently complex, that a fully implemented system will always fall in the latter category. Yet there is little doubt that the current system could be made less complex, and that this would both reduce bugs, improve verifiability, and lower the barriers of entry for the members of the public wishing to inspect, review, and analyse the system.

3.3.1 Documentation

The electronic voting project has published a lot of excellent high-level documentation, covering architecture, voting processes, cryptography, and so forth. This is available at https://brukerveiledning.valg.no/Dokumentasjon/default.aspx. In contrast, we found the low-level documentation of source code to be somewhat thin on the ground. This started with unsuccessfully trying to compile the source code according to the supplied build guide[3], and the little source code documentation that exists did not always serve to clarify the purpose and requirements of a particular package, method or class.

A key challenge for someone outside the project trying to understand the code, is that while there are several diagrams showing the architecture *logical* relationships between the different election actors (VCS, RCG, et al.) and election processes, it is not really self-evident how they relate to the actual source code, and how the different source code modules relate to each other. This is compounded by the fact that most source code comments and Javadoc are quite thin, and generally focus on *what* the code does, rather than *how* it fits into the overall architecture and processes.

For example, there are a large amount of interdependencies between the evoting, counting and vsframework modules, and all of them build on classes from the protocol module. Yet it has not been possible to find any kind of documentation defining the logical relationships between the different projects, and how they are split.

Better documentation could be expected to simplify further development, maintenance and refactoring of the system, and might also help promote code reuse. In addition to this, mnemonic believes that improving the state of the source code documentation would be likely to advance the stated project goal of transparency by making it somewhat easier for members of the public to understand and review the code.

An additional challenge observed during the review is that there is often a lack of clear distinction between classes that are live production code, and classes that are legacy remnants, proof of concept code, or part of a test harness. In a number of places, the review

[3] Downloading the source code and dependency tarballs and following https://brukerveiledning.valg.no/Dokumentasjon/Dokumentasjon/Build_guide_V0.5.pdf was not initially successful; additional manual configuration of the Maven build tool was required

has found code that uses insecure settings, but after some investigation does not appear to be part of the "live" election system.

As a final example, consider the two classes `CredentialsGenerator` and `CredentialsGeneratorJCE`, which can be found in the `com.scytl.evote.protocol.integration.voting.impl` package. The classes are nearly identical, defining the same methods, and containing the same comments. Neither class is a subclass of the other. The following comment is the only overall description of what the classes do:

```
/**
 * Cryptographic component used to generate the voter credentials.
 */
```

To someone who is not closely familiar with the entire code base, several immediate questions are raised:

- Why are the classes different?
- What are the technical differences?
- When are they used?
- Where are they used?
- Which voter credentials are we talking about?

A bit of investigation indicates that the `CredentialsGenerator` class is used by the cleansing classes in the counting module, as well as by the evoting module, while `CredentialsGeneratorJCE` is used by the administrative interface in vsframework to generate a voter's individual keys. However, without access to additional low-level documentation, it is quite time-consuming to understand fully the place of these classes, and their relationship with each other.

Recommendation.

- Create or improve low-level documentation, describing the relationships between the source code packages, and how they are fit into the overall system architecture

3.3.2 Security levels and encapsulation

From a cryptographic point of view, there would be few things nicer to review than code using standard algorithms and primitives with internally consistent defaults, exposed via a common high-level API. Ideally, most regular application code should not be required to manage keys and other cryptographic objects; quoting the security expert Thomas H. Ptacek, "*if you're typing the letters A-E-S into your code, you're doing it wrong*".

The point being made by Ptacek is that the implementation and use of cryptography is extremely difficult to get right, and that software developers should not be required or expected to have that kind of specific domain knowledge. Software produced is likely to be

much safer if cryptographic functionality is systematically delegated to a (carefully reviewed and tested) library with a simple API and secure default settings.

The Internet voting system uses a large number of different cryptographic algorithms and parameters, and they are used pervasively throughout much of the code base. Algorithm names (such as AES or SHA256) tend to be hardcoded into each class, and the (sometimes inconsistent) choices made lead to somewhat variable security levels[4][5]. Looking at specific algorithms, the general picture is as follows:

- Asymmetric algorithms such as RSA and El Gamal are consistently specified with 2048 bit keys. These are roughly equivalent to 112-bit security.
- For symmetric encryption, the Advanced Encryption Standard (AES) is preferred, and used both with 128 and 256 bit keys (usually the latter). This provides 128 and 256 bits of security against brute force attack. Triple-DES (3-DES, DESede) is used as a legacy replacement for AES in a few places. This uses a 192 bit key, but provides only about 112 bits of security due to a meet-in-the-middle attack.
- As a secure hash function, SHA-256 is generally used. This provides 128 bits of security against collision (if an attacker can control parts of the input data), and 256 bits otherwise. However, SHA-512 is used in a couple of places, providing twice the security level. There is also one place in the code where MD5 is used, which seems inadvisable, as it is not considered secure.
- As a message authentication code, HMACSHA256 is generally used. This provides 256 bit security against forgery.
- Random 100-bit passwords are generated in a few places, particularly to protect PKCS #12 keystores. The password hashing used with PKCS #12 is expected to be sufficiently slow, that this will provide roughly 128 bits of security; a brute force attack on these password does not appear to be any easier than factoring the RSA keys.

Cryptographic controls are typically bypassed, and not broken outright. However, it is interesting to note that the "weakest" cryptographic components are actually the public RSA and El Gamal keys, which are used heavily in the core voting protocol. Though 112 bits is thought to be infeasible to break with today's algorithms and computers, and thus secure, it provides a rather slim security margin if improved attacks are discovered.

In the first half of 2013, there have been announced several significant theoretical advances on solving the discrete logarithm problem in certain group structures, via improvements to the index calculus algorithm. This is the mathematical foundation underpinning the security of Diffie-Hellman and El Gamal, and is also quite closely related to the security of RSA. The

[4] For a good meta-survey of studies comparing the effective key lengths of different cryptographic algorithms, see http://www.keylength.com/

[5] The security level of a cryptographic algorithm is usually specified in bits, with the phrase "k-bit" security indicating that the cost needed to defeat the security is on the order of 2^k computations

new results have been published in a series of research papers by Antoine Joux, Razvan Barbalescu, and others[6].

These developments currently only apply to groups of low characteristic, which are not used by the e-voting system. However, there is a distinct (though small) possibility that better algorithms for breaking Diffie-Hellman in $\mathbf{Z}_p$ may also be forthcoming. If this were to happen, it may become necessary to quickly move to significantly longer keys, or from groups over $\mathbf{Z}_p$ to elliptic curve groups, to maintain security. However, the latter would require significant changes to the current e-voting implementation.

Recommendations.

- Review algorithm choices, to ensure that appropriate security levels are used
- Ensure agility, e.g. that cryptographic primitives are parametrised and can be replaced with reasonable effort if necessary

3.3.3 Code duplication

Inconsistency also manifests itself in the fact that the same algorithms are not defined in one place, but specified repeatedly. This can both lead to subtle bugs and inconsistencies, and make it harder to modify the code later on, for instance to replace a cryptographic primitive.

The hard-coded string "SHA256" appears in nearly 40 different classes, but in a few places SHA512 is used instead. Decryption interfaces using the RSA algorithm is implemented in at least four different places (jbasis, protocol, com.scytl.crypto, and in the auditing framework), and six different Base64 libraries are referenced (four from third party libraries, one in jbasis, and one in secure logger; the latter two are versions 2.3.1 and 2.1.1 of the same open-source implementation).

Finally, in the `com.scytl.evote.protocol.signers` package, five different variants of Schnorr signatures can be found, with a high degree of overlap. In principle, a Schnorr signature is a rather simple mathematical object, defined through four external parameters; an El Gamal group (defined in our case by two primes p and q, and a generator element g), a private key (represented as a single integer), data to be signed serialized in a canonical format (e.g. as an octet stream), and a source of randomness.

Instead, the different signature variants implemented contain significant duplication of code, but different external interfaces, and different semantic assumptions on what type of data the input represents.

```
DecryptionPrivateKeySigner:
```

[6] The most recent paper, by Barbulescu, Gaudry, Joux and Thomé: "A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic", http://arxiv.org/abs/1306.4244

```
      public SchnorrSignature sign(
            final BigInteger[] privateKeys,
            final ElGamalEncryptionValues encVoteOptValues,
            final BigInteger[] decVoteOptIDs,
            final BigInteger[] publicKeys)

PartialDecryptionExponentSigner:
      public SchnorrSignature sign(
            final SecretVoterExponent secVoterExp,
            final ElGamalEncryptionValues encVoteOptValues,
            final PartialReEncryptionParameters partReEncVoteOptIDs,
            final PublicVoterParameter pubVoterParam,
            final BigInteger svcID
            final CryptoFactory cryptoFactory)

PartialDecryptionPrivateKeySigner:
      public SchnorrSignature sign(
            final BigInteger[] partReEncPrivKeys,
            final PartialReEncryptionParameters partReEncVoteOptIDs,
            final BigInteger[] partReEncPubKeys,
            final BigInteger svcID,
            final CryptoFactory cryptoFactory)


ReEncryptionFactorSigner:
      public SchnorrSignature sign(
            final BigInteger[] reEncExponents,
            final ElGamalEncryptionValues[] encVoteOptValues,
            final ElGamalEncryptionValues[] reEncVoteOptValues,
            final BigInteger[] publicKeys)

VoteOptionSigner:
      public SchnorrSignature sign(
            final EncryptionExponent exponent,
            final ElGamalEncryptionValues encVoteOptValues,
            final BigInteger[] publicKeys,
            final BigInteger voterID)
```

Table 2. Schnorr signature variants in the signer package

The result of this is five different classes that implement the whole Schnorr primitive, containing a large amount of partly duplicated code. From both a maintainability and auditability point of view, it would probably be better to have one "core" Schnorr implementation containing the cryptographic code, and smaller adapters (as required) that encapsulate the various business logic surrounding it, such as whether it is a vote option or a partial decryption exponent that should be signed.

Recommendations.

- Identify security functionality implemented in multiple places, and consolidate / refactor the affected code
- Consider increased separation between cryptographic logic and application / protocol logic, to simplify validation

3.3.4 Dead code

The difficulty of distinguishing "live" production code from other artifacts in the source code repository has previously been alluded to, and it is appropriate to give some examples of code in the main source code repository which is actually not used in production.

That some of the code in jbasis library is unused by the e-voting code is perhaps not unreasonable, since it is shared between multiple projects. There are however plenty of other examples of unused code. Perhaps the most obvious are the `settlement` packages in the counting project, as this is not handled as part of Internet voting.

A more tricky example is all code which was used exclusively by the Java voting client (running in the browser), since this has later been replaced with a Javascript implementation. This probably includes parts of the cryptographic code in the protocol project, but it can be difficult to tell which parts.

It can also be quite difficult to distinguish classes that are part of test harnesses for specific environments (e.g. system and integration testing), and which contain actual production code. A rule of thumb is that the former contain hard-coded configuration, but this is not always clear cut. A particularly odd instance is the `HardcodedElectionManagementService` class, which contains several test passwords, but is also inherited from by the production class, `RemoteElectionManagementServiceImpl`.

There are also a number of free-standing tools and utilities that appear not to be for use in the production environment, but these are fairly simple to identify. This category includes the `tpm-central-tool` in the auditing project, and the `cast-vote-robot-tool` in vsframework.

Finally, there are a few places where unimplemented or buggy functionality is simply commented out, or otherwise disabled, often making the remaining code redundant. This is particularly common in jbasis, but also occurs elsewhere.

Recommendations.

- Indicate more clearly whether code is to be used in test or production, e.g. via Javadoc
- Remove unused code from the main repository

3.4 Automated testing

An early step taken to get better acquainted with the code base, was by running automated tools for static analysis. In particular, FindBugs[7] with the "Find Security Bugs"-plugin, as well as PMD[8], were used. These are freely available tools that are simple to set up and run, which use heuristic techniques to discover possible or probable code errors. While this type of general source code review in general is beside the scope of this audit, it has been a useful technique to discover possible problem areas and catch low-hanging fruit. In particular, FindBugs has been quite helpful.

Using an hour to run FindBugs revealed a surprisingly large number of "hits", i.e. possible bugs in the code. While many of these are almost certainly false alerts, and some are minor errors that have no impact on correctness or security, there are also apparent bugs that may have greater consequences. An example of code that was flagged by FindBugs is the following snippet:

```
com.scytl.evote.auditing.asyslog.SecureSyslogAppender#initialize():
      byte[] lastEncSessionKey = null;
      SecretKey secretKey = null;
      PrivateKey userPrivateKey = cf.getUserPrivateKey(_pkcs12Certificate,
            _pkcs12Password.toCharArray());
      if (lastEncSessionKey != null) { … code omitted … }
      final LoggingEventProcessorList loggingEventProcessor =
          LoggingEventSyslogProcessorListFactory
              .createLoggingEventProcessorList(
                  cf.getUserPublicKey(_pkcs12Certificate),
                  userPrivateKey, _numberLogLines, this,
                  _timerLogMilliseconds, secretKey);
```

Table 3. Example of code flagged by FindBugs: strange handling of null values in code

The reader may note that `lastEncSessionKey` always will be null when it is checked, so the if-clause looks unlikely to execute. Because of this, the `secretKey` parameter will remain unset until it is passed as an argument to `createLoggingEventProcessorList()`[9]. What the developer actually intends to happen is hard to understand from the code, at least in isolation.

Another useful set of findings came from the "Find Security Bugs" plugin, which among other things looks for insecure sockets, old cryptographic algorithms, and weak random number generator. Most of the time, `java.security.SecureRandom` is used to provide strong randomness, but there are a number of places where it is not.

[7] http://findbugs.sourceforge.net/ and http://h3xstream.github.io/find-sec-bugs/
[8] http://pmd.sourceforge.net/
[9] It appears that this parameter may not actually be used for anything by the `createLoggingEventProcessorList` method.

```
com.scytl.evote.protocol.integration.voting.RCGCrypto:
      public RCGCrypto(final CryptoProvider cryptoProvider,
              final ElectionManagementService kms) {
          super(cryptoProvider, kms);
          _random = new Random(new Date().getTime());
          _rcManager = new ReturnCodeManager();
      }
```

Table 4. Example code flagged by Find Security Bugs plugin: why is a "Crypto" class using an insecure random generator?

In this case, it is not immediately clear why the Return Code Generator crypto class should be using an insecure random number generator. It has thus been looked into more closely, with the analysis being presented in 0 and 4.2.6.2.

Some of the error categories that recur frequently in the FindBugs reports are the following:

1. Code paths that (may) allow a parameter to be null when evaluated, causing a `nullPointerException` if it occurs. Manual analysis on a case-by-case basis is needed to judge whether this will happen in practice, and whether this may impact the correctness or security of the code.
2. Direct conversions between `String` and `byte[]` that don't take encoding into account. This will usually be fine as long as the locale and encoding is fixed (e.g. when exchanging data between different systems), but may otherwise cause strange compatibility errors. There may also be impacts relating to how data is processed (e.g. whether multi-byte characters are fully processed as input to a hash). We observe that a few conversions are locale-aware, but that most are not.
3. Matters concerning thread safety, synchronization, and methods that manipulate static objects.
4. Situations where references to internal objects are exposed, or references to external objects are used in protected contexts.
5. Dead code, and other "strange" code constructs which may not work as intended.
6. Return values from certain system calls, notably methods such as `java.io.File#delete`, are sometimes unchecked. These calls do not throw exceptions if they fail, and may thus fail silently.

It is highly recommended to use static analysis tools (whether open-source or commercial) as an integrated part of the *general* quality control regime in the e-voting project. This works as a good supplement to regular unit testing, and can be integrated in the build process in a similar way. More than one tool can be used in tandem, as they often implement different heuristics and catch different types of flaws.

Domain-specific static analysis tools may also be helpful for *specific* purposes, such as uncovering security flaws early in the development lifecycle. There exist multiple dedicated commercial tools for static security analysis, though they have not been used as part of this audit.

In a system such as e-voting, which presumably has particularly stringent requirements of correctness and overall quality, it seems strange that such simple aids are apparently not in use. Static analysis has also been carried out by third-party researchers in the past, yielding similar results, which makes it even more surprising that it has not yet been adopted.

Recommendations.

- Use one or more static analysis tools as an aid to improve code quality
- Integrate such tools in the regular build and test process

3.5 On random-number generation

Quoting the American mathematician Robert Coveyou, "*Random Number Generation is too important to be left to chance*". Indeed, high-quality randomness that is cryptographically secure, is of utmost importance for nearly all the other cryptographic algorithms used for electronic voting. Because of this, it is an obvious area that an adversary may try to attack.

The insecure `java.util.Random` generator is referred to by about a dozen classes in the e-voting codebase, and all of these have been inspected manually. A few of these occurrences are worth a closer mention, though most of the use appears innocuous:

- `com.scytl.evote.protocol.integration.voting.RCGCrypto`, as previously mentioned, uses an insecurely seeded (and thus predictable) `Random` instance to select which zero knowledge proofs from the VCS to verify. Checking only a fraction of the zero-knowledge proofs is acceptable because the probability of successful cheating decreases exponentially with the number of votes, and has been analysed previously. However, a cheating VCS might try to predict the RCG random seed, in order to guess which proofs will be checked. The probability of successful cheating will still remain low, but it may be prudent to use a secure random generator here.
- `com.scytl.evote.vsframework.client.commands.kms.PrintVotingCardCommandsUtil`: An insecure `Random` instance is used in the `generatePrimeNumbers` method, which, confusingly, generates random nine-digit integers that are not tested for primality. This does not seem to have a security impact, but at the very least contradicts the method name and source code comments.
- `com.scytl.jbasis.math.BigNumber` takes a `Random` instance which may or may not be secure as part of its constructor, and as a parameter to the `safePrime` method. Thus the caller must be responsible for passing an appropriately strong random generator at all times; a `BigNumber` returned from the `safePrime` method may be "safe" in a purely technical sense, yet useless for cryptography.
- `com.scytl.jbasis.util.StringUtils` takes a `Random` instance as input to utility methods such as `randomString`, with the same implications as for the previous class.
- `com.scytl.evote.evoting.votingclient.Randomizer` is an insecure randomizer, but does not appear to be in use.

- `com.scytl.evote.protocol.integration.voting.parser.model.EmlElectoralModel` uses an insecure random generator to shuffle the party lists. This is probably acceptable; the order of the party lists should be randomly selected, but there does not *appear* to be a secrecy requirement related to this.

For the rest of the code base, when randomness is needed, `java.security.SecureRandom` is used. This is a good choice. However, we observe that the default constructor for `SecureRandom` is used throughout (with a *single* exception).

It does not appear that any settings for the random generator is specified via configuration, which means that the actual "Secure Random" generator instance utilized is likely to be implementation dependent, with defaults that will vary between platforms and providers. On a Linux system, the default provider would normally be expected as Sun's `NativePRNG` utilizing `/dev/urandom` as the underlying entropy source, but this is hard to verify when it is not specified explicitly.

Recommendations.

- Review use of insecure pseudorandom generator `java.util.Random`
- Make specification of `SecureRandom` algorithm and provider explicit, through `SecureRandom#getInstance` or via configuration

4 Review of low-level security implementation

This chapter covers technical analysis of low-level security and crypto functionality in the e-voting system. Much of the basic functionality is implemented by Scytl in the `com.scytl.jbasis`-library, but only some parts of this is used by the Norwegian e-voting code. In the e-voting project itself, much of the cryptographic code is located in the `com.scytl.evote.protocol` project. However, there are also a number other classes scattered through the other parts of the code base. We also look into the `com.scytl.slogger` (secure logger) library, which is independent of the e-voting code and can be evaluated by itself. Finally, we consider how the cryptographic code is used key generation.

4.1 Analysis of com.scytl.jbasis

The jbasis project consists of 19 packages, implementing basic Java functionality. This covers low-level cryptographic interfaces, smart card token handling, XML and utility classes. The project appears to be fairly old, and belonging to a common codebase that Scytl utilizes in multiple settings, not only for electronic voting in Norway.

As a result of this, the project contains a significant amount of code that is either retained for legacy compatibility, or provides functionality which does not appear to be in use by the e-voting system. For instance, the classes used to support IAIK as a Java Cryptographic Extension (JCE) provider are not relevant to our purposes. We will try to limit our analysis to those parts of jbasis that are in use by the Norwegian e-voting system.

A central part of the jbasis code are the crypto packages, which provide abstract interfaces to low-level cryptographic functionality, as well as a concrete instantiation built on the third party BouncyCastle library. Flaws in this part of the code may lead to significant and pervasive vulnerabilities.

4.1.1 com.scytl.jbasis.crypto

The package offers a lot of cryptographic functionality, though mainly in the form of abstract interfaces, which are realized in the `com.scytl.jbasis.crypto.bc` package. However, there are also some central definitions here.

4.1.1.1 CryptoFactory.java

CryptoFactory follows the *Abstract Factory* design pattern and generally only defines the interface. However, it also defines a number of constants.

```
SYMMETRIC_CIPHER = "DESede/ECB/PKCS5Padding";
DESEDE_CBC_SYMMETRIC_CIPHER = "DESede/CBC/PKCS5Padding";
AES_SYMMETRIC_CIPHER = "AES/ECB/PKCS5Padding";
AES_CBC_SYMMETRIC_CIPHER = "AES/CBC/PKCS5Padding";
SIGNATURE_ALGORITHM = "SHA1withRSA";
SECRET_KEY_ALGORITHM = "DESede";
AES_SECRET_KEY_ALGORITHM = "AES";
```

```
PBE_SECRET_KEY_ALGORITHM = "PBEWithMD5AndDES";
SECRET_KEY_LENGTH = 24;
HASH_METHOD = "SHA1";
HMAC_HASH_METHOD = "HMACSHA1";
KEYPAIR_ALGORITHM = "RSA";
PUBLIC_KEY_LENGTH = 2048;
PUBLIC_KEY_LENGTH_PROPERTY = "jbasis.public.key.length";
PBE_ITERATIONS = 100;
PBE_SALT = {(byte) 0xA9, (byte) 0x9B, (byte) 0xC8, (byte) 0x32,
(byte) 0x56, (byte) 0x35, (byte) 0xE3, (byte) 0x03 };
```

There are a couple of strange things to take note of here:

- Triple-DES is used as the default cipher, while AES (Advanced Encryption Standard) must be specified explicitly. Because the default mode is ECB (electronic codebook), a different mode must be specified if one wants to encrypt more than one block of data (64 bits with 3-DES, 128 bits with AES).
- The default settings for password-based encryption, specified by the parameters PBE_SECRET_KEY_ALGORITHM, PBE_ITERATIONS and PBE_SALT, are not secure. Investigation indicates that these PBE settings are not actually used by the Norwegian e-voting system, but it may be a concern for other users of the library.

The implication of these defaults is that unsuitable cryptographic parameters may be chosen when a `CryptoFactory` is used directly with default settings. Motivation for these parameter choices appears to be backwards compatibility. This seems somewhat fragile, as a developer using the jbasis library will have to consistently specify appropriate algorithms, rather than using secure defaults.

While it does not seem like these defaults are used to a significant extent by the e-voting code, we shall keep it in mind when we look at the library usage moving forward.

4.1.1.2 Other classes

Most of the package just provides the generic interfaces, but there are a few other classes with actual content.

- `AsymetricKeySize` [sic] defines the constants `KEY_SIZE_1024` and `KEY_SIZE_2048`.
- `CA.java` encapsulates basic Certification Authority behaviour, and is used to handle certification requests. Certificates are issued with serial numbers generated with `java.util.Random`, which means that these are potentially predictable. This is not a problem.
- `CryptoEncoding` converts between DER encoding and PEM format.
- `CryptoUtils` contains two methods to compare public and private keys, even if the keys are issued by different crypto providers. These only check that the (RSA) exponent and modulus are equal, and will not work with other ciphers. It also contains a method for signature verification (which does not appear to be in use) and the skeleton of a method for hashing all the files in a directory (which is not implemented).

All of this seems reasonable. Other classes in the package are either simple interface definitions, or do not appear to be in use by Norwegian e-voting. The latter category includes the classes `CryptoVerifier`, `FastBitCommittment`, `FastBitCommitmentVerifier`, and `HashAdapter`.

4.1.2 com.scytl.jbasis.crypto.bc

This package contains an actual crypto instantiation, using the BouncyCastle JCE provider. External entry points are primarily through `BCCryptoFactory`, `BCPrivateKey`, `BCPublicKey`, and `BCSecretKey`.

4.1.2.1 BCCryptoFactory

The `BCCryptoFactory` class is just a very thin wrapper around the other BC-classes, which instantiates these as needed. For secure random number generation it uses a private final instance of `java.security.SecureRandom`. This behavior generally seems secure.

Note that the methods `createSecretKeyFactory` and `createSecretKeyWithPassword` will create password-based encryption schemes with the insecure PBE defaults from the base `CryptoFactory` class.

4.1.2.2 BCPrivateKey and BCPublicKey

The `BCPrivateKey` class implements an RSA private key object, with a straightforward constructor (based on a modulus and a private exponent) and utility classes. There are also methods for reconstructing a `BCPrivateKey` object based on an input stream or a byte array, but these are heavily commented out. Comments indicate that this code may not be stable, and it appears not to be in use.

The corresponding `BCPublicKey` class contains the same functionality, but for *public* RSA keys instead.

4.1.2.3 BCSecretKey

The `BCSecretKey` class provides various ways to produce a secret key object. The essential thing to note here is that calling `BCSecretKey` without any arguments will generate keys suitable for the default algorithm, which is triple-DES ("DESede").

Thus it is necessary to keep an eye on how these are initiated. Providing a `KeySpec` or a `SecretKeySpec` as input will be necessary to get keys that are suitable for e.g. AES.

4.1.2.4 Other classes

The remainder of the classes mainly implement their generic interfaces by acting as wrappers for the BouncyCastle JCE provider.

Going through the interfaces class by class it can be easy to overlook errors. Most of the interfaces with BouncyCastle appear to be correct. However, we did observe one bug in the `BCPKCS7Envelope` class, which makes the following incorrect call to the provider:

```
CMSEnvelopedDataGenerator.generate(new CMSProcessableByteArray(data),
      CMSEnvelopedDataGenerator.DES_EDE3_CBC, 128, "BC");
```

Bug.

The `org.bouncycastle.cms.CMSEnvelopedDataGenerator` class provides the following method signature:

```
generate(CMSProcessable content, java.lang.String encryptionOID,
      int keySize, java.lang.String provider)
```

However, triple-DES must be used a 192 bit key.

4.1.3 com.scytl.jbasis.math

The math package contains the `BigNumber` class, along with smaller supporting classes for polynomial evaluation and modular computation. This is used in connection with Shamir Secret Sharing (see **Error! Reference source not found.**).

4.1.3.1 BigNumber

The `BigNumber` class is a decorator wrapping a regular `BigInteger`, with a few extra methods to support modular polynomial arithmetic. We shall comment on those features which are actual extensions.

- The value `PRIME_CONFIDENCE = 100` indicates that probable primes should have a probability of at least $1 - 2^{100}$ of actually being prime. This is consistent with the default settings for `BigInteger.probablePrime()`.
- The constructor `BigNumber(final long bitLength, final Random rnd)` casts `bitLength` from long to int, truncating in the process. This seems unlikely to be a problem, as actual bit lengths are not expected to exceed a few thousand.
- The `higherPrime()` method works by generating a strong random prime of the same order of magnitude as the input, adding it to the input value, and finally searching for the next higher prime. This appears to be a secure approach, as it avoids any obvious correlation with the starting value.
- The `modPowAlt()` and `modPowCrt()` methods attempt to provide optimized replacements for `BigInteger.modPow()`, that is to say, efficient methods of computing the expression "x^y *modulo* n". Note that Chinese Remainder Theorem-based exponentiation may leak information about the factorization of an underlying RSA modulus, through the time it takes to execute.
- The `safePrime()` method computes a prime $p = 2q+1$ such that p and q are both prime.

The class appears secure, provided that `SecureRandom` is used as a generator whenever a secure `BigNumber` is needed. This is only done explicitly in the `higherPrime()` method; in other cases a `Random` object is passed as a parameter. Care should also be taken with

using the custom exponentiation methods, though these are apparently not in use by the Norwegian e-voting project.

4.1.4 com.scytl.jbasis.shares

The shares package implements Shamir's Secret Sharing algorithm[10], along with associated helper classes for persisting shares. The idea behind the algorithm is to represent a secret string as a random *(k-1)*'th degree polynomial over a finite field, and distribute n shares representing points. Given threshold knowledge of *k* unique points, the polynomial (and thus the secret) can be reconstructed by interpolation, whereas *k-1* shares will yield no information at all.

The `com.scytl.jbasis.eraser` package is used extensively to ensure that garbage collection is performed whenever secret data should be erased from memory. This is a somewhat arcane topic, and we have not performed a detailed analysis of the garbage collection scheme itself. However, while the `Eraser` class is used extensively throughout the electronic voting system, the impression is that it may not be used consistently.

The high-level interface for the package is `ShareAlgorithm`, which is implemented by `ShamirAlgorithm`. The `ShareManager` class is used as a bridge between the Shamir scheme, and a variety of connectors for securely persisting the shares, e.g. to a deck of smart cards.

4.1.4.1 ShamirAlgorithm

The high level Shamir algorithm implementation is fairly straightforward. The two main methods are `split` and `merge`, which interface with `ShamirShadowManager` to do the actual work.

```
split: String secret, int shares, int threshold, (optional int modulus)
    1. Encode the secret as a BigNumber
    2. Set up a new ShamirShadowManager for the parameters.
    3. Invoke ShamirShadowManager.marshall()
    4. Return a Vector of shares
```

```
recover: Vector shares
    1. Set up a new ShamirShadowManager to recompute the secret
    2. Invoke the getter method secret() to recover the actual value
    3. Convert it back to a String
```

There is also a `merge` method, which combines two (presumably sub-threshold) vectors of shares into a single vector.

4.1.4.2 ShamirShadowManager

The `ShamirShadowManager` class is used to implement the actual secret splitting and merging. We consider the two processes in turn.

```
ShamirShadowManager: constructor (splitting):
    1. Check the input parameters (shares, threshold, modulus if present)
```

[10] Invented by Adi Shamir, see http://en.wikipedia.org/wiki/Shamir's_Secret_Sharing

```
    2. Generate a random LagrangePolynomial of degree threshold-1
    3. Evaluate the polynomial at 1, 2, … to obtain the share values
ShamirShadowManager.marshall():
    1. Iterate through the list of points
    2. Compute the String representation of each share
    3. Output the vector containing all the shares
```

Moving on to the process of merging the shares, we have the following pseudocode, which looks correct.

```
ShamirShadowManager: constructor (merging):
    1. Add each share to the vector containing the points
    2. Verify that all the parameters are consistent
    3. Verify that we have enough shares
    4. Reconstruct the LagrangePolynomial
    5. Evaluate it at 0 to obtain the secret
```

Vulnerability.

The code to generate the random Lagrange polynomial in the constructors for splitting selects polynomial coefficients from the wrong distribution, which violates the theoretical security proof for Shamir Secret Sharing.

Recall that the polynomial coefficients must be randomly chosen from the set {0, 1, …, modulus – 1). In the constructor, the code sets the `BigNumber` m equal to modulus – 1, and proceeds to compute the coefficients as follows:

```
From the com.scytl.jbasis.shares.ShamirShadowManager constructor:
      new BigNumber(m.getBitLength(), new SecureRandom(m.toByteArray())));
```

Consider the two methods being called here.

- The `getBitLength()` method wraps `BigInteger.bitLength()`, which (for positive values) "*Returns the number of bits […] in the ordinary binary representation.*"
- The `BigNumber` constructor wraps `BigInteger (int numBits, Random rnd)`, which "*Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to (2^{numBits} - 1), inclusive*".

To see why the code is incorrect, assume for example that the prime `modulus` is 19, which means that coefficients should be selected uniformly at random from the range [0, 18]. The value of `m` is 18 and the `m.getBitLength` will be 5. However, the random `BigNumber` coefficient that is subsequently generated will be uniformly selected from the range [0, 31] (e.g. 2^5-1).

The polynomial mathematics classes will perform the requisite reductions (mod 19), meaning that the subsequent computations will be correct despite coefficients being out of range. However, the bug means that the Shamir polynomial coefficients will be selected from the wrong statistical distribution, that is no longer uniform.

Impact and recommendation.

It looks as though this bug will cause the theoretical security proof for Shamir's Secret Sharing scheme to fail. Initial analysis indicates that practical exploitation of the issue is expected to be hard, but could conceivably be attempted by an adversary with access to several shares (yet fewer than the threshold). We have not had time to perform a detailed impact analysis.

The issue has been communicated to Scytl, who have acknowledged the finding as a bug.

4.1.4.3 Low-level persistence classes

The `ShareManager` class works as a link between the secret sharing implementation, and the low-level routines for persisting the shares. These include `SCConnection` and `SCConnector`[11], which define the interfaces, and the supporting utility classes `ConnectionType` and `ConnectorOptions`. There are also specific implementations:

- `FileConnection / FileConnector` – for writing to local file
- `TokenConnection / TokenConnector` – for writing to an actual smart card

Writing secret shares to a local file with `FileConnection` must *only* be used for testing, as this implementation appears to use the insecure default Password-Based Encryption settings from the Jbasis crypto package, described in Section 4.1.1.1. There may also be other security requirements enforcing that key shares from production must only be persisted to a secure storage medium.

Regarding the token classes, we note that the `TokenConnection.readShare` and `writeShare` methods use triple-DES (`CryptoFactory.SECRET_KEY_ALGORITHM`) to encrypt data on the tokens, rather than AES. This may not be consistent with e-voting project requirements, though it is likely to be secure.

Apart from this we have not found anything remarkable in these classes.

4.2 Analysis of com.scytl.evote.protocol

The protocol-3.2.5 project contains implementations of central cryptographic functionality used by the Internet voting protocol. In many cases this is complementary to the jbasis crypto code, other times it works as a replacement. The degree of overlap is not clearly documented, and when we look at some of the more complex implementations, we shall have to keep track of which library is used where.

4.2.1 com.scytl.evote.protocol.ciphers

The ciphers package implements five different basic cipher types, an encrypted envelope, and a partial decryptor for ElGamal votes.

4.2.1.1 AsymmetricCipher

The `AsymmetricCipher` class provides basic asymmetric encryption functionality, using as default `RSA/ECB/PKCS1Padding` and the standard SunJCE crypto provider. Alternate

[11] SC stands for "Share Card" in this context

constructors lets the caller specify the algorithm and/or provider manually. The suitable cipher object is retrieved from the JCE provider via `Cipher#getInstance`. This seems safe, provided that PKCS#1 v1.5 RSA encryption is what you want.

Additional methods are provided to encrypt and decrypt, either byte arrays of serializable objects. The implementations of `encrypt` and `decrypt` seems reasonably straightforward.

It is interesting to note that `getBlockSize()` is called throughout to check whether a block cipher is in use, as this should never be the case for a regular asymmetric cipher such as RSA. The additional code to handle this case seems to add a bit fair of unnecessary complexity, and it is unclear what value it brings.

It is worth noting that there is also an `AsymmetricCipher` class in jbasis, and that these may easily be confused.

4.2.1.2 ElGamalCipher

The `ElGamalCipher`[12] provides access to the `BlockElGamalEngine`, which is defined in the `com.scytl.evote.protocol.engine` package. It implements the same mechanisms as AsymmetricCipher to encrypt and decrypt either serialized objects or byte streams.

As opposed to the previous class, the inner `encrypt(final byte[] bytes, final BigInteger publicKey)` and `decrypt` methods explicitly handle input messages that are longer than a single block.

```
encrypt: byte[] bytes, BigInteger publicKey:
   1. Obtain input / output block sizes from the BlockElGamalEngine
   2. Compute the number of blocks, and the remaining space
   3. Pad the bytes with zeroes to a block boundary
   4. Use the BlockElGamalEngine to encrypt the blocks one by one
   5. Wrap the encrypted data and the number of zero bytes (remainder) in a
      BlockElGamalEncryptedData object
```

```
decrypt: EncryptedData encData, BigInteger privateKey:
   1. Unwrap the encrypted data and remainder from the input
   2. Obtain the input / output block sizes
   3. Use the BlockElGamalEngine to decrypt the blocks one by one
   4. Output the decrypted byte array
```

This seems straightforward, and should maintain confidentiality as long as the core `BlockElGamalEngine` is implemented securely. Note that El Gamal's encryption scheme without a secure padding scheme is malleable; in particular it is not secure if the adversary is able to decrypt chosen ciphertexts. Because of this, the algorithm is not suitable for all applications.

4.2.1.3 JCEEnvelope

In the JCE terminology, an *envelope* consists of a message encrypted with a random symmetric key, with the key being encrypted with an asymmetric mechanism. This is in the

[12] Named after Taher El Gamal, see http://en.wikipedia.org/wiki/ElGamal_encryption

style of Cryptographic Message Syntax[13]. Encryption schemes of this type are also commonly referred to as "hybrid encryption", with the terms "Key Encapsulation Mechanism" (KEM) and "Data Encapsulation Mechanism" (DEM) being used in the provable security literature to refer to the asymmetric and symmetric components. Several KEM/DEM schemes (with KEMs based on e.g. RSA and ElGamal) are standardised in ISO 18033-2.

The `JCEEnvelope` class implemented by Scytl builds an envelope using `AES256/CBC/PKCS5Padding` as the data encapsulation mechanism, and a user-supplied `CryptoFactory` which is used to provide an `AsymmetricCipher` (4.2.1.1) instance for the key encapsulation. This uses PKCS #1 v1.5 padding as default; if replaced with OAEP padding this the scheme would be similar to the RSAES-KEM defined in ISO 18033-2.

```
createEnvelopedData:
      CryptoFactory factory, PublicKey pubkey, byte[] plaintext:
   1. Generate a random secret AES key
   2. Construct a random IV using SecureRandom
   3. Encrypt the plaintext with the AES key
   4. Get an AsymmetricCipher object from the factory
   5. Encrypt the AES key and IV with the AsymmetricCipher and the pubkey
```

```
readEnvelopedData: CryptoFactory factory, PrivateKey pubkey,
      byte[] encKey, byte[] encIv, byte[] encryptedData:
   1. Get an AsymmetricCipher object from the factory
   2. Decrypt the secret AES key
   3. Decrypt the random IV
   4. Decrypt the data
```

There are a few features of the `JCEEnvelope` class that can be contrasted with the standard hybrid encryption schemes.

- It is not necessary to encrypt the random initialization vector, this does not give any security benefit.
- The standardized schemes in ISO 18033-2 schemes use an additional Message Authentication Code (MAC) together with the symmetric encryption, to ensure the integrity of the symmetric ciphertext. This is an important part of the strong security proofs for the hybrid schemes.
- Using a hybrid construction without a MAC means that the JCEEnvelope will almost certainly not satisfy the strong IND-CCA2 notion of security. Using `AsymmetricCipher` default settings, it is likely to satisfy the weaker IND-CPA ("semantic security") notion.
- Note that the `AsymmetricCipher` instance is from `com.scytl.jbasis.crypto` (4.1.1) and not the package implementation (4.2.1.1)
- The security of the enveloped scheme is completely dependent on the externally specified factory instance and its `AsymmetricCipher` instance being securely initiated

[13] Defined in RFC 3369, based on the earlier PKCS #7

One should consider implementing one of the ISO schemes for enveloped data in this situation. Adding a MAC would prevent e.g. bit-flipping attacks on the symmetric ciphertext.

4.2.1.4 MessageCipher

The `MessageCipher` class can be used to encrypt a `String` with the `BlockElGamalEngine`. The class appears identical to `ElGamalCipher`, apart from working on a `String` rather than a `byte[]`. It is unclear why this is implemented as a separate class.

4.2.1.5 SymmetricCipher

The `SymmetricCipher` class is similar to the `AsymmetricCipher` class in 4.2.1.1, but using `AES/CBC/PKCS5Padding` and a minimum key length of 128 bits. Again, the code makes systematic checks of `_cipher.getBlockSize`, this time to determine whether it use an initialisation vector (IV).

Since `_cipher.getBlockSize` will be nonzero whenever a block cipher is used, this does not quite make sense. For instance, if a stream cipher were to be specified, `getBlockSize` should be zero, and no IV will be used. This is likely to be insecure, since it seems to imply keystream reuse if encrypting multiple times under a fixed key. As far as we can tell, the ciphers used by e-voting will all be block ciphers and follow the secure code path. However, this once again seems like a very fragile state of affairs.

4.2.1.6 VoteOptionCipher

The `VoteOptionCipher` class acts as an interface to (re-)encrypt or decrypt a series of vote options with El Gamal, using the underlying `VoteOptionElGamalEngine` (see 4.2.3.2). The main feature of this class is that it encrypts a large vector of vote options, using a similarly-sized vector of public keys, and a fixed secret exponent which is reused for all the options.

This is a somewhat unusual El Gamal optimization, as it would normally be very dangerous to reuse the secret exponents. However, provided that there are no simple algebraic relationships between the vote options or public keys, there does not appear to be any "trivial" attacks on the scheme. It is assumed that the e-voting project has performed a thorough cryptographic analysis before making this protocol optimization.

Re-encryption is done by multiplying the old ciphertext with another "round" of of El Gamal:

- $phi_{new} = phi_{old} * publicKey^{exponent} \pmod p$
- $gamma_{new} = gamma_{old} * generator^{exponent} \pmod p$

It is somewhat difficult to consider the security of this computation without looking more carefully at the protocol context in which it is used, but there are no obvious weaknesses in this.

4.2.1.7 VoteOptionPartialDecryptor

The `VoteOptionPartialDecryptor` class is used to partially (re-)encrypt or re-decrypt encrypted vote options, using the underlying `VoteOptionPartialDecryptorEngine` class. This is a different kind of re-encryption than in the previous sections:

- $param_1 = phi_{old}^{exponent}$
- $param_2 = gamma_{old}^{privateKey}$
- $param_3 = param_1 * param_2$
- $gamma_{new} = gamma_{old}^{exponent}$

Once again there are no obvious weaknesses in this computation, but it is somewhat difficult to evaluate the security without more careful evaluation of the protocol context in which it is used, since it differs from the usual "textbook" El Gamal scheme.

4.2.2 com.scytl.evote.protocol.digests

There is only one class in this package, `HashFunction`.

4.2.2.1 HashFunction

The `HashFunction` class is a simple implementation to provide access to a cryptographic hash function through the JCE provider. By default, SHA-256 is used. This is a good default choice for a secure cryptographic hash function. The UTF-8 input encoding is explicitly specified when converting the input string to a byte array, meaning that the encoding will be consistent across platforms. This seems very reasonable.

4.2.3 com.scytl.evote.protocol.engine

The engine package contains three different special-purpose ElGamal implementations.

4.2.3.1 BlockElGamalEngine

The `BlockElGamalEngine` is conceptually fairly straightforward. It is used to encrypt an array of bytes by dividing it into blocks and applying El Gamal encryption to each block. It needs to be initialized with a secure El Gamal key, e.g. a triple *(p, q, g)* where *q* and *p = 2q+1* are prime, and *g* is a generator of the group. The following pseudocode is used to encrypt / decrypt a single block.

```
encryptBlock: byte[] in, int inOff, int inLen, BigInteger publicKey:
   1. Check that the input length is appropriate
   2. Represent the input array as a BigInteger
   3. Generate a random exponent k which is not equal to -1, 0, 1, nor
      greater than the group order p
   4. Compute the ciphertext tuple (gamma, phi) from input, k and publicKey
   5. Convert it back to a byte array and return
```

```
decryptBlock: byte[] in, int inoff, intLen, BigInteger privatekey:
   1. Check that the input length is appropriate
   2. Split the input array in two, and convert to two BigIntegers
   3. Decrypt using the private key and the El Gamal identity
   4. Convert to byte array
   5. Do post-processing (e.g. add leading zeroes) and return
```

This appears correct for encrypting individual blocks, under the usual preconditions for secure El Gamal encryption, i.e. as long as an adversary is not able to decrypt chosen ciphertexts, and random exponents k are never re-used.

Since El Gamal is malleable and no chaining mode is defined, there is no implied relation between blocks. Unless the integrity of the overall ciphertexts is maintained (using a MAC or a signature), an adversary would be free to shuffle, swap or modify blocks in a longer string. This may be a potential vulnerability, depending on the usage.

4.2.3.2 VoteOptionElGamalEngine and VoteOptionPartialDecryptorEngine

The `VoteOptionElGamalEngine` is used to (re-)encrypt and decrypt vote options. The vote options are represented as `BigInteger` instances, and the class simply performs modular arithmetic on these values according to the definition of El Gamal. All parameters (e.g. keys and exponents) are passed as external arguments.

The final engine class, `VoteOptionPartialDecryptorEngine`, is used for partial encryption and decryption of vote options. This is again reasonably straightforward modular arithmetic, based on the definition of how the computation should be carried out.

In both cases, the implementations look straightforward, but usage (accessed via the `VoteOptionCipher` and `VoteOptionPartialDecryptor` ciphers) should be checked against the protocol definitions.

4.2.4 com.scytl.evote.protocol.generators

The generators package includes functionality to generate Diffie-Hellman and El Gamal keys.

4.2.4.1 DHKeyGeneratorHelper and DHParametersHelper

This pair of classes contains necessary functionality to produce a Diffie-Hellman key.

The `DHParametersHelper` class is used to pick a Diffie-Hellman group. The relevant methods are:

- `generateSafePrimes`: Find a prime q and safe prime $p = 2q+1$, of a specified bit length and with specified certainty (of the primeness of p and q)
- `selectgenerator`: Find a generator g the subgroup of order q

Both methods appear correct. They use `SecureRandom` for their entropy.

Meanwhile, `DHKeyGeneratorHelper` implements the following methods:

- `calculatePrivate`: produces a random private key of specified length
- `calculatePublic`: derive the corresponding public key using the

Once again `SecureRandom` is used as the random generator.

4.2.4.2 ElGamalKeyPairGenerator and ElGamalParametersGenerator

This pair of classes uses the aforementioned Diffie-Hellman helpers to produce ElGamal keys. The implementations appear to be quite straightforward, without any complex logic.

4.2.5 com.scytl.evote.protocol.integration.mixing.base

The "integration mixing base" implements zero-knowledge proofs for vote mixing, building on other protocol classes, including `VoteOptionCipher` (4.2.1.6) and the various `SchnorrSignature` variants (4.2.9).

4.2.5.1 ReEncryptionProofs, ReEncryptionProofGenerator and ReEncryptionVerifier

These classes are used to represent, generate and verify Schnorr signatures on secret exponents that are used for re-encryption at mix-nodes. To generate a proof on a vote group collection, each vote group is signed using the secret exponents that were used to re-encrypt it, with the `ReEncryptionFactorSigner` class from the signer package (4.2.9.4). This is done for each vote group in the collection.

4.2.5.2 VoteGroupManager

While not implementing much cryptographic functionality as such, the `VoteGroupManager` class is notable for being the only place in the e-voting codebase where the underlying random number generation (RNG) algorithm and provider for use with `SecureRandom` is explicitly specified in the source: "`SHA1PRNG`" and "`SUN`". In all other cases observed, the default constructor is used.

Explicitly specifying the algorithm and provider is actually quite good practice, since it ensures that one does not obtain a default value with unexpected properties. For the purposes of the shuffling that is done in the class, the choices are perfectly appropriate. Indeed, `SHA1PRNG` is often the default `SecureRandom` algorithm, particularly on Windows machines.

4.2.5.3 VoteMixer and VoteMixingIOValidator

The function of vote mixing is to validate, shuffle and re-encrypt votes as part of a mix network. It is perhaps worth noting that a regular `AsymmetricSigner` (4.2.9.1) is used for this purpose, rather than the various El Gamal signers used elsewhere in the package. Re-encryption is done using the `VoteCipher` class, and shuffling is done with `java.util.Collections#shuffle` utilizing `SecureRandom` for strong randomness.

4.2.6 com.scytl.evote.protocol.integration.voting

The voting package and its sub-packages contain a fair bit of crypto protocol implementation for integration between the different voting components. A central element is the `ElectionManagementService` interface, which specifies 88 (!) core methods for election management.

4.2.6.1 BaseCrypto

The `BaseCrypto` class is a base class that is extended by several of the component-specific crypto classes. It contains a number of default methods to verify the integrity of a vote. It specifies the use of UTF-8 encoding, and SHA-256 as a hash algorithm. The following security-relevant methods are implemented:

- `verifyEncryptVoteSig` checks that the JSON vote data contained in a `VoteBean` object has a valid RSA signature, and that the certificate is consistent with the voter's Common Name, authentication time, and the election identifier.
- `verifyAuthenticationToken` uses the Authentication Service certificate to check an user's authentication token.
- `verifyVoteZKProof` uses the `VoteOptionSigner` (4.2.9.6) class to verify a zero-knowledge proof (Schnorr signature) protecting the vote options contained in an encrypted vote.
- `createVoteHash` computes a SHA-256 hash over a `VoteBean` object, an authentication token and a timestamp.

The two first methods use an `ElectionManagementService` instantiation to perform the actual verification. `BaseCrypto` is extended by the `ConfigCrypto`, `PrinterCrypto`, `RCGCrypto` and `VCSCrypto` classes. The latter two implement additional verification that should be analysed.

4.2.6.2 RCGCrypto

The `RCGCrypto` class contains a few additional crypto methods for the return code generator.

In the `verifyPartialDecryptSig` method, the VCS public key is used to verify a (RSA) signature from an `AsymmetricSigner`, on the contents of a `PartialDecryptBean`.

The `verifyExpZKProof` and `verifyKeyZKProof` methods check (El Gamal) zero-knowledge proofs of knowledge on partially decrypted votes from the VCS, made with the `PartialDecryptionExponentSigner` and `PartialDecryptionPrivateKeySigner` respectively.

The `isOptimized` method is a simple routine that returns true with 12.5% probability. It is used elsewhere to determine whether the two partial decryption ZK proofs shall be verified, as an optimisation to reduce the workload on the RCG. Interestingly, the method uses an insecure `java.math.Random` instance initialized using new `Date().getTime()` as the seed. This seems dubious, since it means that the VCS might try to guess the seed to predict which ballots will be verified by the RCG, since it can guess when the RCG was rebooted. It is unclear how this would affect the security of the protocol, but it would be more conservative to use a `SecureRandom` instance for this purpose.

The final method of interest is `generateReturnCodes`. It contains slightly more logic, and is worth describing in more detail.

```
generateReturnCodes: PartialDecryptBean partialDecrypt,
      VoterIdentifier voterIdentifier, String electionType,
      String electionEventId:
   1. Obtain the RCG symmetric key from the ElectionManagementService
   2. Initialize a VoteOptionPartialDecryptor cipher instance
   3. Use it to partially decrypt the data in the PartialDecryptBean, using
      the RCG's ElGamal key retrieved from the ElectionManagementService
   4. For each of the partial return codes obtained (that are not empty),
      generate a return code using the partial return code, voter ID and
```

```
      symmetric RCG key
   5. The actual return code generation is performed by the
      com.scytl.evote.protocol.managers.rcmanager.ReturnCode class. It uses
      HMACSHA256 to digest the partial return code and voter ID.
```

With the exception of the potentially questionable random numbers generated by the `isOptimized` method, the `RCGCrypto` class appears to be correct.

4.2.6.3 VCSCrypto

The Vote Collection Server also has a dedicated crypto class, serving as the counterpart of `RCGCrypto`. Of particular interest is the `partiallyDecryptVote` method, which is used to produce the `PartialDecryptBean` objects that are processed by the three verification methods in `RCGCrypto`.

```
partiallyDecryptVote: VoteBean vote, VoterIdentifier voterIdentifier:
   1. Obtain the VCS symmetric key from the ElectionManagementService
   2. Initialize a VoteOptionPartialDecryptor cipher instance
   3. Generate a secret voter exponent for re-encryption, which is derived
      (via the keyManager) by encrypting the voter ID with the VCS
      symmetric key
   4. Re-encrypt the encrypted votes using the secret voter exponents and
      the concatenation of the VCS ElGamal keys
   5. Generate the corresponding public parameters used for partial
      decryption
   6. Generate the two zero-knowledge proofs, for knowledge of the secret
      voter exponent and the VCS ElGamal private keys
   7. Digitally sign the partially re-encrypted vote and the two zero-
      knowledge proofs, using the VCS RSA key.
   8. Wrap everything into a PartialDecryptBean
```

There is another variant of the method, `partiallyDecryptVoteWithoutSigning`, which proceeds identically but skips step 7. It does not seem to be in active use.

The `VCSCrypto` class contains a large number of other methods, but these appear to perform comparatively simple signature and signature verification calls, using the appropriate access methods.

4.2.6.4 ElectionManagementService and its implementations

The `ElectionManagementService` class specifies a central interface collecting all the central functions needed to set up an election, and is used *extensively* by the other classes.

There is an immediate implementation of the interface in `HardcodedElectionManagementService` containing hard-coded credentials and configuration, which can be used directly for testing. The actual implementation used for production should be `RemoteElectionManagementServiceImpl` from the `com.scytl.evote.evoting.vcscommon.service` package.

However, there is a rather strange class inheritance relationship at work here: the `RemoteElectionManagementServiceImpl` does not only implement the interface general, it *also* extends the `HardcodedElectionManagementService` class. With the tremendous

amount of methods defined, it is somewhat tricky to *verify* that the "REMSI" implementation in fact correctly overrides all those methods that use hard-coded settings, and that the methods that are not overridden are all safe. Moreover, there are a few classes which explicitly instantiate a `Hardcoded` service object, including `BaseCrypto`.

While there have not been any obvious implementation bugs observed, this state of affairs seems unnecessarily tangled and fragile.

4.2.6.5 CredentialsGeneratorJCE

The `CredentialsGeneratorJCE` class uses the jbasis package to generate a key pair, a certificate signing request, and finally a signed certificate in PKCS #12 representation, wrapped in a `CredentialBeanJCE`. This is used to provide a dedicated key pair for each voter.

A Common Name for each voter certificate is computed using a SHA-256 hash of the voter ID, which means that they are opaque identifiers, but not secret. The certificates are signed using the default algorithm from jbasis, i.e. SHA1withRSA.

This appears reasonably straightforward.

4.2.7 com.scytl.evote.protocol.managers.keymanager

The `KeyManager` class in this package is used as an invoker to perform all key-related operations, i.e. creation, storage and retrieval of any kind of cryptographic key material. It does not implement any logic of its own. However, some of the other classes in the package do. Much of the content here supplements or replaces that of the `jbasis` library, and one should be careful as to which classes are actually used.

Many of the classes are simply provider interfaces that define simple defaults and interfaces to the underlying providers. In this case we do not give a detailed description of each one, but note the relevant default parameters found.

`CertAuth`	`DIGITAL_SIGNATURE_ALGORITHM_DEFAULT = "SHA256withRSA";`
`CertificateAuthority`	`SIGNATURE_ALGORITHM = "SHA256withRSA";`

`Certificate`
`CertificateSigningRequest`
	`CERTIFICATE_TYPE = "X509";`
	`DIGITAL_SIGNATURE_ALGORITHM_DEFAULT = "SHA256withRSA";`
	`PROVIDER_NAME_DEFAULT = "SunRsaSign";`
	`CHARACTER_ENCODING = "UTF-8";`

`GenerateKeyPair`	`KEY_PAIR_GENERATION_ALGORITHM_DEFAULT = "RSA";` `KEY_BIT_SIZE_DEFAULT = 2048;`
`GenerateSecretKey`	`KEY_GENERATION_ALGORITHM_DEFAULT = "AES";` `KEY_BIT_SIZE_DEFAULT = 256;`
`GenerateKeyStore`	`KEY_STORE_TYPE_DEFAULT = "JCEKS";` `PROVIDER_NAME_DEFAULT = "SunJCE";`
`PKCS12`	`PROVIDER_NAME_DEFAULT = "SunJSSE";` `KEY_STORE_TYPE = "PKCS12";`

	`CHARACTER_ENCODING = "UTF-8";`
`PublicVoterParameter`	`DEFAULT_PROVIDER_NAME = "SunJCE";`
`SecretVoterExponent`	`SYMMETRIC_CIPHER_ALGORITHM = "AES/CBC/PKCS5Padding";` `DEFAULT_PROVIDER_NAME = "SunJCE";`
`SymmetricKey`	`KEY_GENERATION_ALGORITHM_DEFAULT = "AES";`

In this case, the options selected are reasonably self-consistent, in the sense that secure choices of SHA-2, AES and RSA are consistently favoured as default algorithms. It remains inconsistent in the sense that the defaults offer different security levels:

- AES-256 provides roughly 256 bits of security
- RSA-2048 offers ~112 bits of security[14] against factoring, a similar level as 3-DES
- SHA-256 offers ~128 bits of security against collisions, a similar level as AES-128

With the chosen parameters sizes (and the similar choice of 2048-bit keys for El Gamal, though this is not specified separately in these classes), the public-key algorithms are likely to be the least resistant against brute-force attacks. Given current knowledge, even 112 bits of security will not be feasible to attack in the short and medium term, but the security margin with respect to future breakthroughs is lower for RSA and El Gamal than for the other algorithms.

4.2.7.1 ElGamalShareManager, RSAShareManager

The `ElGamalShareManager` and `RSAShareManager` classes are specific extensions of the `com.scytl.jbasis.shares.ShareManager` class (4.1.4.3) used to split and reconstruct keys with Shamir's secret sharing algorithm. The former is the most complex adaptation, as it splits an *array* of El Gamal keys into an array of split shares. Both classes also override the `saveNext` and `readNext` methods for interfacing with smart card storage, though it is not obvious from the code alone why this is needed.

4.2.8 com.scytl.evote.protocol.managers.rcmanager

Similarly to the previous package, the `rcmanager` classes are used to manage return codes. Once again the `ReturnCodeManager` class itself just provides the entry point, and the actual logic is implemented in the various classes. We will have a closer look at the cryptographic defaults specified, as well as specific classes.

Class	**Parameters**

`AnswerCode`
`EmptyVoteCode`
`PartyCode`
`PositionCode`
`PreferenceCode`
`ReturnCode`
	`HMAC_ALGORITHM = "HMACSHA256";`
	`The MAX_BIT_SIZE of the codes is 14.`

[14] Following the NIST SP 800-57 recommendations from 2012, see http://csrc.nist.gov/groups/ST/toolkit/key_management.html

`BallotIdentifier`	`HMAC_KEY_BIT_SIZE = 256;` `HMAC_ALGORITHM = "HMACSHA256";`

`DecryptedCodeColl`
`DecryptedCode`
`EncryptedCodeColl`
`EncryptedCode`
	`SECRET_KEY_BIT_SIZE = 256;`

`HashedReturnCode`	`HASH_ALGORITHM = "SHA256";` `TRUNCATED_DIGEST_BIT_SIZE = 160;`

The various Code-classes all use variants of HMAC as a message authentication code, with SHA-256 as the hash. This is a primitive that provides a very high security margin (roughly 256 bits). Similarly, the encryption classes use 256 bit keys and invoke the default parameters from `com.scytl.evote.protocol.ciphers.SymmetricCipher` (4.2.1.5), which should result in `AES/CBC/PKCS5Padding` being used as the default algorithm.

The HMAC key length is only specified in the `BallotIdentifier` class. In the other cases, the size will also be 256 bits, but the value is specified under the hood in `com.scytl.evote.protocol.managers.keymanager.GenerateSecretKey` and not explicitly in the code.

4.2.8.1 *AnswerCode, EmptyVoteCode, PartyCode, PositionCode, PreferenceCode, and ReturnCode*

These classes are all nearly identical. Each class is used to generate a particular type of return code, with the `generate()` method being the key component for this.

The significance of the various *_MAX_BIT_SIZE = 14 of the various classes is that a return code shall be exactly 14 bits of length, i.e. selected from the range [0, 8191]. Why this particular range was chosen (rather than e.g. [0, 9999]) is not obvious.

Every class generates its return code in the following manner:

1. Convert the code identifier (e.g. "Answer ID", "Empty Vote Constant", "Party ID", etc.) and the voter identifier to a byte array.
2. Use the Return Code Generator's (arbitrary) symmetric key to generate a `SecretKey` for the HMAC.
3. Compute a HMAC over the byte array from (1) using the secret key.
4. Convert the HMAC digest to a BigInteger, and truncate it modulo $2^{MAX\_BIT\_SIZE - 1}$
5. Return the resulting short integer.

This will give statistically random return codes that are related to the hashed identifiers and the RCG secret key in a deterministic way. Of course, since the codes are truncated to short values, it is trivially possible to generate a "valid" code (with probability $1:2^{14}$ of success) using brute force guessing. This is presumably irrelevant for the protocol, as the probability of multiple correct guesses quickly becomes negligible.

4.2.8.2 HashedReturnCode

The `HashedReturnCode` class is also "strange", in the sense that it makes a seemingly arbitrary transformation that does not make sense without further context. In short, it takes a `ReturnCode` object, hashes it using SHA-256, then truncates the output to 160 bits. This is a somewhat nonobvious, though secure, way to produce a random-looking 20-byte message digest.

We note that the collision resistance of this construction is down to 80 bits, due to the birthday paradox applied to the truncated hash. This means that a brute force attack to find two `ReturnCode` objects with the same hash (given sufficient degrees of freedom) might be feasible in theory, for a well-funded and patient adversary.

The overall security margin for this construction is thus not very high. On the other hand we expect that spending massive computational effort to find two return codes that hash to the same value is unlikely to be a cost-effective approach for an attacker wanting to harm the e-voting system.

4.2.9 com.scytl.evote.protocol.signers

Corresponding to the ciphers package (4.2.1), this is where the local implementations of signature algorithms are located. The package contains a remarkable number of slightly different implementations of Schnorr signatures, with minor differences in purpose and method signatures. While all the implementations look superficially correct for their particular use cases, this is noticeably difficult to verify by hand.

4.2.9.1 AsymmetricSigner

The `AsymmetricSigner` class is the only signer which is not a custom implementation based on El Gamal and Schnorr's signature scheme, instead defaulting to the SunRsaSign JCE provider to utilize the `SHA256withRSA` algorithm specification. This is expected to be secure as long as the RSA modulus is of at least 2048 bits.

The `sign` and `verify` operations are specified for both byte arrays and serializable objects, and simply wrap the `update` and `sign`/`verify` methods from the provider.

4.2.9.2 DecryptionPrivateKeySigner

The class uses a Schnorr[15] signature as a zero-knowledge proof that the signer knows the private key used to decrypt a set of vote option identifiers. The SHA-256 algorithm is used as a hash. Main cryptographic logic is implemented in the `update`, `sign` and `verify` methods.

```
update: ElGamalEncryptionValues encVoteOptValues,
      BigInteger[] decVoteOptIDs, BigInteger publicKey,
      MessageDigest messageDigest:
   1. Represent all the decVoteOptIDs as one large input byte string
   2. Compute the product of all the decVoteOptIDs (modulo p)
   3. Compute the product of the phi values from encVoteOptValues
```

[15] Named after Claus-Peter Schnorr, see http://en.wikipedia.org/wiki/Schnorr_signature

```
   4. Hash the decVoteOptID byte array, and represent it as a BigInteger
   5. Compute phi and gamma for the zero-knowledge proof as either:
         a. gammaZKP = encVoteOptValues.getGamma(),
            phiZKP = phi-product * decVoteID-product^-1
         b. gammaZKP = encVoteOptValues.getGamma()^-1,
            phiZKP = decVoteID-product * phi-product^-1
   6. Compute the derived generator and public key for the proof:
         a. _generatorZKP = _generator * gammaZKP^hash (modulo p)
         b. _publicKeyZKP = _publickey * phiZKP^hash (modulo p)
```

```
sign: BigInteger[] privateKeys, ElGamalEncryptionValues encVoteOptValues,
      BigInteger[] decVoteOptIDs, BigInteger[] publicKeys
   1. Initialize a SHA-256 message digest
   2. Compute a representative of the publicKeys as the product of all the
      keys in the array (modulo p)
   3. Call DecryptionPrivateKeySigner.update()
   4. Securely generate the randomness for the Schnorr signature, in the
      range [2, q-1], and exponentiate it
   5. Hash the randomness, group generator, and the public key
      representative (which is the data to be signed)
   6. Compute the private key part of the signature using the privateKeys
      array, the randomness from step 4, and the hash from step 5
```

```
verify: SchnorrSignature sig, ElGamalEncryptionValues encVoteOptValues,
      BigInteger[] decVoteOptIDs, BigInteger[] publicKeys:
   1. Repeat the 3 first steps of the signing procedure
   2. Recompute the exponentiated randomness from the signature
   3. Recompute the hash from the randomness, generator and public keys
   4. Verify that the result is the same as in the signature
```

This implementation of Schnorr signatures appears correct. There are three minor observations regarding the `sign` method. Neither has a significant security impact. These observations are common to all the Schnorr signature methods in this package.

Observations.

- The randomness is generated in the range [2, q-1], rather than [2, q-2] in the BlockElGamalEngine. In practice choosing *q-1* as the exponent would be a poor choice, but the probability of generating that value with SecureRandom is in any case negligible, so there is not really any point in checking.
- In the original description and security proofs of Schnorr's signature scheme, the hash function should output uniformly random elements of $\mathbf{Z}_p$ rather than n-bit strings. However, a paper by Neven, Smart and Warinschi[16] from 2009 indicates that this should be fine as long as the hash function output length is at least twice the security level. Thus, using SHA-256 without any further processing should be sufficient for 128-bit security.
- If 256-bit security is desirable, SHA-512 should be used.

[16] *Hash Function Requirements for Schnorr Signatures*, http://www.neven.org/papers/schnorr.html

4.2.9.3 PartialDecryptionSigner and subclasses

The `PartialDecryptionSigner` class, together with subclasses `PartialDecryptionExponentSigner` and `PartialDecryptionPrivateKeySigner`, are used to provide zero-knowledge proofs on secret keys used for partial decryption. Functionally this is quite similar to the previous signer, but with a somewhat different API and different data input.

In `PartialDecryptionSigner`, we find the following utility methods:

```
computeCoeficients: int numBetas, CryptoFactory cryptoFactory,
      MessageDigest messageDigest:
   • Generates numBetas pseudorandom coefficients, by iterating SHA-256
   • The initial seed is produced from the passed messageDigest, so this
     object must contain some data when the function is called
```

```
generateBeta: BigInteger alpha1, BigInteger alpha2,
      CryptoFactory cryptoFactory, MessageDigest messageDigest:
   • Generate a single pseudorandom coefficient, by invoking messageDigest
     (likely SHA-256) on the alphas, and hashing the result again with
     SHA-256
```

```
generateRandomNumber: BigInteger q:
   • Generate a secure random number in the range [2, q-1]
```

The motivation for these utility methods is completely opaque from the source code alone – no rationale or context is provided, and an external description of the algorithms is necessary to understand why any of this is needed or useful. The `generateRandomNumber` method is identical to the inline code in `DecryptionPrivateKeySigner`, and again arbitrarily chooses the range *[2, q-1]*, rather than *[2, q-2]* which is used in other places.

In the two subclasses, actual signing and verification is implemented. These proceed more or less identically, as below:

```
PartialDecryptionPrivateKeySigner#sign():
   1. Verify that input parameters are consistent
   2. Initialize a SHA-256 hash with the public inputs
   3. Call computeCoeficients [sic] using the initialized hash
   4. Derive a private key representative, based on an array of input keys,
      and the pseudorandom beta coefficients
   5. Generate randomness for use in the Schnorr signature
   6. Compute alphas from randomness and the data to be signed
   7. Call PartialDecryptionSigner#generateBeta and use beta as the hash in
      the Schnorr signature
   8. Compute the remainder of the Schnorr signature using the derived
      private key
```

Arithmetically, these implementations appear correct, in the sense that a Schnorr signature is produced on the `alpha` values using a derived private key. The alphas and keys are computed differently in the two cases, but the implementation logic is very similar.

4.2.9.4 ReEncryptionFactorSigner

This is yet another re-implementation of Schnorr signatures for a particular use-case. In this case, the private key for signing is generated from the array of re-encryption exponents (`reEncExponents`) that is used as input. The data that is signed, includes the hash of the set of public keys used to encrypt vote option identifiers. The effective generator used for the signature is computed by hashing the encrypted and re-encrypted votes.

Once again the core Schnorr signature primitive looks secure, but based on the source code a bit of thinking is needed to understand what it actually signs.

```
sign: BigInteger[] reEncExponents, ElGamalEncryptionValues[]
      encVoteOptValues, ElGamalEncryptionValues[] reEncVoteOptValues,
      BigInteger[] publicKeys
   1. Compute the following BigInteger products (all modulo p):
         a. publicKey = product of elements of publicKeys []
         b. gammaProd1 = product of gammas in encVoteOptValues
         c. gammaProd2 = product of gammas in reEncVoteOptValues
         d. phiProd1 = product of phis in encVoteOptValues
         e. phiProd2 = product of phis in reEncVoteOptValues
   2. Compute the message as the hash of the products, as well as the
      ElGamal group generator
   3. Derive an offset generator: generatorZKP = generator * publicKey^message
   4. Sign the publicKey product of terms, using the derived generator as a
      base, and the reEncExponents as the private key
```

Thus it is in fact the set of public keys that are signed with the re-encryption exponents.

4.2.9.5 SchnorrSignature, SchnorrRandomExponent and SchnorrRandomExponentPool

These are simple container classes for signature objects and exponents, and do not contain cryptographic logic. The two latter classes are of unclear benefit; `SchnorrRandomExponent` seems only to be used by `VoteOptionSigner`, and `SchnorrRandomExponentPool` is passed around as an argument in the `PartialDecryptionSigner` classes, but does not seem to be used.

4.2.9.6 VoteOptionSigner

This is the fifth variation over a Schnorr signature implementation, to fit yet another specific use-case. As such, much of the code is identical to the previously analysed incarnations, with minor variations to trip up the unwary reader.

The most obvious differences from the previous implementations is that the `com.scytl.evote.protocol.integration.eraser.Eraser` is used more aggressively to ensure that secret information is safely garbage collected, and that the data to be hashed is hex encoded for compatibility with JavaScript.

The class appears to implement a secure Schnorr signature, but with the many competing implementations and slightly different method APIs throughout, one has to be very careful to use the right algorithm for the application context.

4.2.10 com.scytl.evote.protocol.tools

The "protocol tools" packages contain a number of free-standing command-line tools. It is unclear how these are used, but they should be checked.

4.2.10.1 ParametersGenerator

The `ParametersGenerator` tool is used to generate El Gamal parameters q and p = 2q+1, based on the desired bitlength and the certainty required for primality testing.

The actual p's and q's are generated via a call to `ElGamalParametersGenerator` from the generators package (4.2.4.2), and the `ParametersGenerator` performs a self-test for additional verification that the set of El Gamal parameters are sound.

The default choices for bitlength and certainty are 2048 bits and $1-2^{-100}$, respectively, which is consistent with choices made elsewhere.

4.2.10.2 GeneratorSelector

The `GeneratorSelector` tool is used to pick a generator element, based on group parameters q and p provided as input. Presumably, it runs right after the `ParameterGenerator` tool.

Generators are actually generated via a call to `ElGamalParametersGenerator` (4.2.4.2) from the generators package, and the class makes a self-test for additional verification that the parameters are sound.

The default size of the El Gamal parameters is given by `PARAM_SIZE_DEFAULT = 2048`, which is consistent with other classes. However, the size parameter, whether the default is used, or it is parsed from command line, is not actually used to verify the length of the parameters. In fact, it is only used in the input and output filenames. This is fine if the tool is run in conjunction with `ParameterGenerator`, but in general the code would be more robust if it did not rely on the filenames being correct.

4.2.10.3 PrimalityTester

The `PrimalityTester` tool is simple command-line wrapper for the library function `BigInteger#isProbablePrime`, with user-selectable certainty. The default value is 1000, which is exponentially more certain (and thus significantly slower) than the choice of 100 used elsewhere.

4.2.10.4 PrimeNumberGenerator

The `PrimeNumberGenerator` tool is, somewhat misleadingly, not used to generate primes in itself, but to determine primes that are quadratic residues (or quadratic non-residues) for specified El Gamal parameter sets. These primes are then used as vote option identifiers.

4.3 Analysis of other cryptographic packages

4.3.1 com.scytl.crypto

The `com.scytl.crypto` package is for some reason not found as part of the "Internettstemmegivning" code repository, but the sources can be found in the "Skanning" repository. It is used by the `com.scytl.evote.auditing` and `com.scytl.slogger` packages, to provide "boxed" cryptographic functionalities.

According to source code comments, the aim of the package is to provide a simple pre-configured high level cryptographic API for achieving simple tasks.

4.3.1.1 CryptographicConstants

The `CryptographicConstants` class defines the following set of defaults:

```
public static final String ASYMMETRIC_KEY_ALG = "RSA";
public static final int ASYMMETRIC_KEY_LENGTH = 2048;
public static final String ASYMMETRIC_CIPHER_ALG = "RSA/ECB/PKCS1Padding";
public static final String ASYMMETRIC_SIGNATURE_ALG = "SHA256withRSA";
public static final String SYMMETRIC_KEY_ALG = "DESede";
public static final String SYMMETRIC_CIPHER_ALG =
"DESede/CBC/PKCS5Padding";
public static final String HMAC_ALG = "HMACSHA256";
public static final String HASH_ALG = "SHA-256";
public static final String KEYSTORE_TYPE = "pkcs12";
public static final String CERTIFICATE_TYPE = "X.509";
public static final String CERT_CHAIN_BUILDER_TYPE = "PKIX";
public static final String SECURE_RANDOM_ALG = "SHA1PRNG";
public static final int SECURE_RANDOM_SEED_LENGTH = 20;
```

This can be compared with similar default parameters specified in other places, notably the jbasis (4.1.1.1) and protocol (4.2.7, 4.2.8) packages.

The most notable choice in the above list, is probably the use of 3-DES as `SYMMETRIC_KEY_ALG` and `SYMMETRIC_CIPHER_ALG`. While 3-DES provides an adequate security level of roughly 112 bits, it is a distinctly less secure option than AES, and also provides significantly poorer software performance.

4.3.1.2 CryptographicAlgorithms

The `CryptographicAlgorithms` class provides a packaged interface to basic primitives: symmetric and asymmetric encryption, digital signature, key generation, random byte generation, a message authentication code, and a hash. The algorithms are specified using the hard-coded identifiers from `CryptographicConstants`.

A feature here that may take some by surprise is that the `generateSymmetricKey` method uses a key generator initialized for use with `SYMMETRIC_KEY_ALG`, i.e. triple-DES. Thus a 192-bit key will be generated. This may not be what is expected if the key is to be used with the (256-bit) message authentication code primitive; the MAC will still be secure, but at the 192-bit security level, rather than at full strength.

4.3.2 com.scytl.evote.vsframework.vscommon.client.utils

4.3.2.1 UtilsRBAC

In the otherwise unassuming `UtilsRBAC` class, we find the method `cipherSymmetrically` which implements password-based encryption. The primary purpose of this methods appears to be decrypting access control tokens used, which are issued externally and used for role-based access control. As such, the implementation has presumably been written to conform with an external specification which may not be under Scytl's direct control. However, it looks like the utility is also used to encrypt data "internally" within the voting system.

This implementation has several advantages over the password-based encryption implementation in the jbasis library, which is insecure with the default algorithm choices. Nevertheless, password-based security is rightly known as a minefield, and the implementation appears to contain significant security flaws. Consider the following source code from lines 111 onwards, containing the meat of the cryptographic logic.

```
String salt =
      "Static salt for use in key genereation while exporting security token";

PBEKeySpec keySpec =
      new PBEKeySpec(password.toCharArray(), salt.getBytes(), 2,
            256);
SecretKeyFactory keyFactory =
      SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
SecretKey key = keyFactory.generateSecret(keySpec);
cipher =
      Cipher.getInstance("AES/CFB/PKCS7PADDING",
            new BouncyCastleProvider());

byte[] iv =
      new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
            0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
            0x0f };

AlgorithmParameterSpec paramSpec = new IvParameterSpec(iv);
cipher.init(mode, key, paramSpec);
byte[] outData = cipher.doFinal(inData);
```

There are three clear security weaknesses in this code:

Vulnerabilities.

1. A fixed IV (initialization vector) combined with AES in CFB (Cipher FeedBack) mode means that the keystream used to encrypt the first block will be identical every time the method is used (with the same password). For input data that shares a prefix, the keystreams and ciphertexts will be identical until they block where they diverge.
   - Given two ciphertexts (protected with the same password), an attacker can obtain

the exclusive-or of the beginning of the plaintexts.
    - Given a plaintext-ciphertext pair (protected with an unknown password), an attacker can decrypt the beginning of other ciphertexts.
2. A fixed IV also means that the same data will encrypt to the same value (under a fixed password) every time.
3. The iteration count for the password hash is set to 2. This is such low value that a brute force decryption attack on the passwords is likely to be feasible, unless passwords are very long.

Two other observations do not have a direct security impact, but may also be worth noting:

4. The use of CFB-mode and PKCS #7 padding with AES is somewhat unusual, though secure as such. It contrasts with the rest of e-voting, which uses CBC-mode and PKCS #5. This seems somewhat arbitrary, but may (as mentioned) be due to an external specification.
5. It is not clear whether the specified password hash will process all 16 bits in the character array holding the password, or only digest the lower 8 bits of each character value. This is not really a problem, as long as the passwords are sufficiently long.

The `cipherSymmetrically` method is used a number of places in the vsframework project, notably the `com.scytl.evote.vsframework.client.commands` packages. For instance, in the method `SymmKeyGenerationCommand#execute` it is used to store the generated symmetric keys to disk using the master password.

Regarding the password hash iteration count, a similar vulnerability was found in the Blackberry backup system in 2010[17]; in that case the value 1 was used as an iteration count. The recommended value when PBKDF2 was proposed in 2000 was 1000 iterations, and the current recommendations are significantly higher due to advancements in hardware speeds.

Mitigation.

1. To fix the IV issues, one needs to handle encryption and decryption modes slightly differently. In encryption mode, a unique random 16 byte IV should be generated and prepended to the output byte stream. In decryption mode, the first 16 bytes of the input byte stream should be used as the IV.
2. The PBKDF2 iteration count should ideally be "as high as possible" without impacting the user experience. Actual values in real-world use tend to range from 10000 and upwards. Ideally, the number should be chosen based on an estimate on how long it should take to crack the password-based encryption, giving current brute-force capabilities and the password policy in use.

Another sound option would be to stick to PKCS #12 as a container format; as this can be used to persist arbitrary data and not just keystores. Writing a simple interface on top of the BouncyCastle PKCS #12 implementation might be a simpler option than creating it from the

[17] See CVE-2010-3741, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3741, and http://www.infoworld.com/print/139436

more general JCE primitives.

4.3.3 com.scytl.evote.counting.ecounting.crypto

The counting package provides yet another small crypto module, consisting of the three classes.

4.3.3.1 TemporalDigestedFile, TemporalDigestedFileInputStream and TemporalDigestedFileOutputStream

A `TemporalDigestedFile` is an absolute path (representing a `File`), together with a digest containing the hash of that file. The other two classes extend `FileInputStream` and `FileOutputStream` to keep track of the message digests as files are read and written.

There is one rather obvious problem with this implementation.

```
Both Stream classes:
      _messageDigest = MessageDigest.getInstance("MD5");
```

Since MD5 is not a collision-resistant hash algorithm, this method may not provide strong guarantees that a `TemporalDigestedFile` has not been actively modified between writing and reading.

Let us consider how this is used. It turns out that the `com.scytl.evote.counting.ecounting.builder.EMLbuilder` class uses `TemporalDigestedFileOutputStream` to write election results data to storage. Meanwhile, the `com.scytl.evote.counting.ecounting.zip.CreateZipFile` class reads the XML files back in with a `TemporalDigestedFileOutputStream`, to create a zip file and export the results.

```
CreateZipFile#createResultsZipFile() excerpt:
      // Read the temporal file
      tdfis =
            new TemporalDigestedFileInputStream(temporalFile);
      byte[] fileContent = IOUtils.toByteArray(tdfis);
      tdfis.close();

      if (!tdfis.checkDigest()) {
            throw new UnexpectedException(
                  THE_TEMPORAL_FILE_HAS_BEEN_TAMPERED
                        + temporalFile);
      }
```

If an attacker has some degrees of freedom relating to how the `EMLbuilder` class writes its output XML files, this is likely to be insecure due to known collision attacks on MD5. Regardless of the attacker capabilities, using MD5 for any kind of secure purpose (with the possible exception of HMAC-MD5) is strongly discouraged and may be in violation of project requirements.

Vulnerability.

• Use of deprecated hash function MD5 to ensure file integrity.
Mitigation. • Replace MD5 with SHA-256, or with a secure MAC.

4.4 Analysis of audit and logging framework

The audit and logging framework consists of the secure-logger (`com.scytl.logger`) and auditing (`com.scytl.evote.auditing.*`) packages. These are reasonable small and self-contained packages, and it stands to reason that the logs produced should be verifiably authentic.

4.4.1 com.scytl.slogger

The secure-logger package extends the log4j library to provide secure logging functionality. What we are particularly interested in here are the core classes, as well as event processors (located in the `com.scytl.logger.processor` sub-packages), which are used to add cryptographic functionality.

4.4.1.1 SecureFileAppender

The `SecureFileAppender` class is the base abstract secure logger class, extending the `FileAppender` class from log4j. It does not actually implement any security features on its own.

4.4.1.2 SecureLoggingEvent

The `SecureLoggingEvent`, extending the log4j `LoggingEvent`, is the basic decorator class for events that are to be logged.

4.4.1.3 LoggingEventProcessor

This is the general interface that all event processors must implement.

4.4.1.4 SignatureEventProcessor

A `SignatureEventProcessor` is used to put a digital signature on a `SecureLogEvent`. A private key for the signature scheme must be specified in the constructor. The public `processEvent` method uses a protected method, `signData`, to create a signature using the `com.scytl.crypto` library. According to the default settings, this will sign with `SHA256withRSA`, which is a secure signature scheme.

A possible issue with the `SignatureEventProcessor` is that it creates the byte array to sign using `String.getBytes`. This encodes the string using the *default* character set, rather than a specific character set such as UTF-8. Hypothetically, this could cause ambiguities if someone is able to modify the default character set, or create a log event containing characters that cannot be represented in the default character set.

4.4.1.5 HMacEventProcessor

A `HMacEventProcessor` is used to produce a message authentication code on a `SecureLogEvent`. The message authentication code is computed over the current event, as well as the *previous* MAC value, which means that the sequence of log entries is preserved. In pseudocode:

```
calculateHMac: LoggingEvent loggingEvent, SecurePatternLayout layout:
   1. Use the layout to create a formatted String from the loggingEvent
   2. Compute the HMAC over the concatenation of the _lastHmac value and
      the formatted event
   3. Update _lastHmac with the computed HMAC value
   4. Base64-encode the HMAC and return it
```

This chaining feature means that intermediate log entries can not be deleted, as that would invalidate HMACs on all subsequent entries. However, a log could still be *truncated*.

The MAC used is `HMACSHA256` as specified in `CryptographicConstants` (4.3.1.1), which is certainly secure. However, the key generated for use with the MAC appears to only be 192 bits in length, since the `generateSymmetricKey` method from the `CryptographicAlgorithms class` (4.3.1.2) is used.

Once again there also seems to be a potential issue with the `String.getBytes` conversion, which does not specify the character set for the conversion.

4.4.1.6 EncryptedHMacEventProcessor

Building on the previous class, the `EncryptedHMacEventProcessor` implements additional `secretKeyLog` functionality, to persist and rotate the HMAC secret keys.

```
secretKeyLog: SecurePatternLayout layout:
   1. Get the existing _lastHMAC and _secretKey, if they exist
   2. Reinitialize the superclass, thus obtaining a new secret key and a
      reinitialized HMAC
   3. Asymmetrically encrypt the new key with RSA
   4. Generate a SecureLoggingEvent containing the _lastHMAC, the old
      secret key, and the encrypted version of the new key
```

By itself, it seems that this would not be secure; an attacker could truncate the key log (and the event log) to hide the key change, and emit new log events using the old MAC key (from the key log). Thus, the class should not be used directly.

4.4.1.7 SignedEncryptedHMacEventProcessor

This is the ultimate event processor class, adding all functionalities. It extends the secret key logger in the `EncryptedHMacEventProcessor` class with an additional `SignatureEventProcessor`, which is used to sign the `secretKeyLog`. The full contents of a `secretKeyLog` event is thus:

- the standard log formatting, including timestamp and log level;
- the current `_lastHMAC` state (before the event was emitted);
- the old HMAC key (if it exists) unencrypted;

- the new HMAC key, encrypted;
- the chained HMAC of the previous four elements (taken with the new key);
- the signature of the previous five elements; and
- a properties object, with the `ENCRYPTED_SESSION_KEY`, `LIBERATED_SESSION_KEY` (if it exists) and `SIGNATURE` flags set.

In the terminology of the implementation, a log "block" consists of the ordered set of log entries made under a single HMAC key. It is clear that the chained HMAC construction makes it infeasible to remove individual events from a block. However, there is nothing here that would prevent an attacker with system access to *truncate* the log.

Potential vulnerability.

If an attacker were to truncate an event log past a block boundary (i.e. key change), she is also able to add new (fake) log events to that block, using the HMAC key that was published at the block change.

This vulnerability should probably be mitigated by operational, rather than cryptographic, means. Our understanding is that the monitoring tool Splunk is used to monitor log events in real time, and that there are configured limits as to how large an event log block will grow before the key is rotated.

The security goal for a secure audit log should, as a minimum, be that any attempt at tampering with the log will be detected (with high probability). Using a real-time log collection tool which resides in a different security zone may satisfy this requirement, if it means that an attacker with local access can not get an opportunity to truncate and modify the log before it is synchronized remotely.

Another potential attack scenario might occur if the attacker can prevent communications between the logging server and the real-time monitoring system for some period of time, and use that window to stop the log generation on the logging server and manipulate the log events before the connection is regained. Because of this, the monitoring should preferably be synchronized in some way.

If there are stronger security requirements related to audit logging, one must consider carefully how this can be achieved within (or without) the secure-logger framework.

4.4.1.8 SignatureEventValidator

A `SignatureEventValidator` is the natural counterpart of the `SignatureEventProcessor`, containing the validation logic for a signed `SecureLoggingEvent`. The `validateEvent` method will take an `event` and a `layout` as input, and check the signature on the event *if* it contains a signature.

An interesting "feature" of the `SignatureEventValidator` is that it succeeds if the event in question does not have the `SIGNATURE` property set.

```
validateEvent: SecureLoggingEvent event, SecurePatternLayout layout:
try {
      if (event.getEventProperties().containsKey(
```

```
            SecureMessageProperties.SIGNATURE)) {
… validate the signature …
      }
}
```

It appears that signed events remain malleable; an attacker could simply remove the `SIGNATURE` property to prevent validation at this point. However, since the `SignatureEventValidator` is not used by itself, we have to look at the HMAC event validators as well.

4.4.1.9 HMacEventValidator, EncryptedHMacEventValidator, SignedEncryptedHMacEventValidator

These classes validate the corresponding processors previously analysed. We will have a closer look at the validation logic for `SignedEncryptedHMacEventValidator`:

```
validateEvent: SecureLoggingEvent event, SecurePatternLayout layout:
      super.validateEvent(event, layout);
      Map<String, String> eventProperties = event.getEventProperties();
      if (eventProperties
            .containsKey(SecureMessageProperties.ENCRYPTED_SESSION_KEY)) {
            if (!eventProperties
                  .containsKey(SecureMessageProperties.SIGNATURE)) {
                  throw new EventValidatorException(
                        "Signature expected: " + layout.format(event));
            }
      }
      _signatureEventValidator.validateEvent(event, layout);
```

The call to super.validateEvent validates the HMAC data. The validation logic then checks if this is a `secretKeyLog` event, in which case the `SIGNATURE` flag is also required to be present, forcing the signature validation to take place.

Potential vulnerability.

However, what happens if someone has removed the `ENCRYPTED_SESSION_KEY` and `SIGNATURE` flags from the log event properties? It appears that the event will validate flawlessly, even though the actual data might be modified. This would not be completely transparent, because it will also prevent the HMAC validator from rekeying with the new key. But it illustrates once more the malleability of the secure logging framework, and the degrees of freedom available to an attacker.

There may be other consistency checks present in the logging and validation framework, though not of a cryptographic nature. Once again, a key countermeasure to ensure the impossibility of log tampering appears to be the presence of the real-time monitoring systems. A different and complementary option, though more intrusive and likely to be less convenient, would be to use some kind of write-once (WORM) storage medium to persist the logs locally on the server that emits them.

4.4.1.10 PublicEncryptedHMacEventValidator and PublicSignedEncryptedHMacEventValidator

The `Public…` classes are used to validate old log blocks, e.g. blocks where the HMAC key has been published due to key rotation. They accumulate a block of events until the HMAC key is "liberated", and then verify them as a batch.

4.4.2 com.scytl.evote.auditing

The auditing package is a small and Norway-specific audit framework built on top of the secure logger package. It does not contain a lot of cryptographic functionality, but we will have a quick look at it regardless.

4.4.2.1 AuditSecureFileAppender

The `AuditSecureFileAppender` class contains its own private extension of `SignatureEventProcessor`, which uses `java.security.Signature` directly, instead of rather than going through the implementation in `com.scytl.crypto` or using e.g. `com.scytl.evote.protocol.signers.AsymmetricSigner` (4.2.9.1).

Presumably there is a reason for this, though it is somewhat unclear what that reason is. The only obvious difference from the boxed `com.scytl.crypto` implementation would be that it is possible to supply an alternate crypto provider, but this is also offered by `AsymmetricSigner`. In any case, the signature implementation appears to be secure as long as a secure private key is used.

Similarly, the class also implements its own routine to decrypt an array of bytes with `RSA/ECB/PKCS1Padding`, supporting multiple providers. In this case, the functionality is already present in `com.scytl.evote.protocol.ciphers.AsymmetricCipher` (4.2.1.1).

The decryption routine makes the same strange mistake as the `AsymmetricCipher` implementation regarding the cipher block size:

```
Decrypt: byte[] what, Key key:
      if (_provider == null) {
            cipher = Cipher.getInstance(RSA_ECB_PKCS1_PADDING);
      } else {
            cipher = Cipher.getInstance(RSA_ECB_PKCS1_PADDING, _provider);
      }
      byte[] toReturn;
      if (cipher.getBlockSize() > 0) {
            byte[][] unpacked =
                  unpackSymmetricMsg(what, cipher.getBlockSize());
            cipher.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(
                  unpacked[0]));
            toReturn = cipher.doFinal(unpacked[1]);
      } else {
            cipher.init(Cipher.DECRYPT_MODE, key);
            toReturn = cipher.doFinal(what);
      }
```

According to the Javadoc for `javax.crypto.Cipher`, `getBlockSize` returns "*the block size (in bytes), or 0 if the underlying algorithm is not a block cipher*". Thus this is never actually supposed to happen.

In general, multi-block encryption with RSA is not advisable, also due to performance reasons. The provider will probably allow one to encrypt (and decrypt) multi-block data, but there is not a chaining mode defined, which means that an attacker would be free to reorder or manipulate the blocks.

Finally, note that the class once again explicitly uses a 192-bit triple-DES key for use with a 256 bit HMAC. This is not in itself insecure, since a k-bit HMAC provides roughly k bits of security, but it seems somewhat strange.

4.4.2.2 RemoteAttestation and RemoteConnector

The `RemoteConnector` class (part of tpm-central-tool, in `com.scytl.evote.auditing.tpm.bizz`) opens an unencrypted remote `Socket` in the aptly named `unsafeConnect` method. However, our understanding is that this tool is a proof of concept implementation that is not actually used in the production system.

4.4.2.3 Other classes

Some of the other classes in the audit package are used to handle keys (for encrypting, signing and/or computing HMACs), but there does not seem to be any significant cryptographic functionality as such.

4.5 Key generation

As the final part of our review, we will look at how keys are generated, using the vsframework command line client utilities. This code is invoked via the `Election_superscript.sh` shell script.

4.5.1 com.scytl.evote.vsframework.client.utils

The utils package contains several utility classes, in particular one that is used extensively during key generation.

4.5.1.1 Utils

The `Utils` class contains a number of useful methods and constants. It defines the following security parameters:

- `P12_PASS_SIZE = 20;`
- `PUK_MIN_LENGTH = 4;`
- `PIN_MIN_LENGTH = 4;`
- `PUK_MAX_LENGTH = 8;`
- `PIN_MAX_LENGTH = 8;`
- `RSA_KEY_SIZE = 2048;`
- `SHARE_PIN = "11111111";`
- `P12_MIN_LENGTH = 8;`

- `P12_MAX_LENGTH = 20;`

Among several methods defined, the following are explicitly relevant for key generation and security.

- `createRandomBase32` creates a random Base32 string of the specified length, typically used as a human-readable password. It does this using a `CryptoFactory` instance passed as a parameter.
- `writePasswordFile` writes a String representing a password, to a file. This file must be well-protected, since it contains the actual secrets.
- The `readPrivateKeyFile`, `writePrivateKeyFile`, and `mergePrivateKeyFile` methods appear to be using the hard-coded `SHARE_PIN` value when dealing with temporal key shares. This is obviously not secure, but it is unclear how much it is used.

4.5.2 com.scytl.evote.vsframework.client.commands.kms

The package contains 44 different command classes that can be invoked via the command line, including key generation. Our focus are the commands that actually generate keys, with reference to the election superscript and Scytl's election configuration guide document. The main functionalities are usually implemented in the `execute` method of the command class. Commands are listed in the order they are first invoked by the election configuration script.

4.5.2.1 InteractiveKms

The `InteractiveKms` is a very large "master" implementation, containing all the commands needed to perform the key setup phase of the voting system. Much of this is not directly cryptographic in nature. However, there are a few points are worth noting.

The signature algorithm property is set to SHA512withRSA, contrasting with the usage of SHA256withRSA elsewhere. This is not insecure, but it is inconsistent.

In the `createPasswordsFiles` method, the method `cipher (final byte[] data)` from `com.scytl.evote.vsframework.client.Interactive` is used repeatedly to process data which is subsequently written to file. This is in fact just a thin wrapper for the method `UtilsRBAC#cipherSymmetrically` (4.3.2.1), which as we have seen contains a number of security weaknesses.

4.5.2.2 CAGenerationCommand

The first step of election setup is to generate the election event certificate authority's root certificate. In the `execute` method, the `keymanager` package (4.2.7) from the protocol project is used to generate a certificate, which is tested for consistency. It is then serialized as a PKCS #12 file, protected with a random password generated by the `Utils` class. The password consists of 20 (`P12_PASS_SIZE`) base32-encoded characters, a total of 100 bits of entropy. This should be OK given the slow PKCS #12 password hash.

Finally, the encrypted PCKS #12 key file is written to disk, and the random password is written to disk as a plain text file, again using the `Utils` class.

With default settings in the `keymanager` package, the certificate generated should be using 2048 bit RSA. Since the master keystore password is written to a cleartext file, care must be taken that it is not accidentally exposed, and that it is securely deleted after use. Measures such as using a dedicated (air-gapped) machine and hard disk encryption will be useful to achieve this.

4.5.2.3 RSAGenerationCommand

After the CA has been generated, public and private RSA keys are generated for each of the system components. The `execute` method loads the CA (reading the password from the console if running interactively), and repeatedly calls `generateKey` for each component. The latter method uses a `keymanager` to generate a public key. This time the key size is specified explicitly as 2048 bits, using the defined constant from the `Utils` class, rather than using the default parameter in the `keymanager` package.

After this, the key is certified using the CA, After this, the code proceeds as in the previous class: the key is tested for consistency, serialized as PKCS #12 and encrypted with a random 20-character base32. The key and password is finally written to disk.

Once again, the approach seems reasonably provided that the password file is kept secure.

4.5.2.4 GenerateCSRCommand

The `GenerateAppCSR` command is used to generate certificate signing requests for the keys that are generated in hardware security modules, rather than by the central KMS. This code should be executed on the various systems connected to a HSM (e.g. the RCG and the VCS).

The `execute` method is fairly long and linear. It initializes a BouncyCastle provider, loads a local keystore from a specified path and with a specified passphrase, or generates a new one if needed. This time, the key algorithm and provider is loaded as properties, and the key size is taken as the default from `keymanager`. This is again slightly different from the two previous classes, though the end result should be the same.

With a keypair at hand, the certificate signing request is invoked via the `keymanager`, in what appears to be a straightforward manner, the request is sanity checked and persisted to a file.

4.5.2.5 GenerateAppCertificateCommand and GenerateCertificateCommand

These classes are used to use the central CA to sign certificates based on the certificate signing requests from the previous point. The `GenerateCertificateCommand` class uses a `BCCryptoFactory` and accesses the BouncyCastle CA provider to certify (via keymanager and jbasis). This appears to be straightforward.

4.5.2.6 AddSymmetricKeyToPKCS12Command

The command is used to add symmetric AES keys to the mix node PKCS #12 files. Key length is taken from the usage focus. The key itself is generated at random via the `KeyManager`. The password is either loaded from the command line, from the password file, or generated

anew. After the key is written to the PKCS #12, it is read back for verification. Once again this seems to be reasonable.

4.5.2.7 SymmKeyGenerationCommand

The `SymmKeyGenerationCommand` is used in step 8 of the generation, to generate a symmetric key which is used by the VCS to generate private voter parameters, and also to provide a symmetric key for the RCG.

The symmetric key is generated via the `KeyManager` and the indicated key length from the focus. At this point, the key is stored using `Utils.encryptFile`, which uses an underlying `JCEEnvelope` to encrypt the file with a certificate, whose path is specified by the mandatory attribute `_certFilePath`.

However, if the optional flag `_storeCipheredSymmetricKey` is specified, the key is *also* persisted to a password-protected file using the infamous `UtilsRbac#cipherSymmetrically` method (4.3.2.1). We observe that this flag *is* set in the `Election_superscript.sh` present in the code base. As previously discussed, the symmetric password-based encryption contains multiple security flaws, particularly if several files are encrypted using the same master passwords.

It appears that the keys are encrypted in this way because they are used by other parts of the configuration script.

4.5.2.8 VoterCredGenerationCommand

The `VoterCredGenerationCommand` is used to generate keys for the individual voters in the electoral roll. It unlocks the CA certificate with the master password, generates a number of threads, and uses the `CredentialsGeneratorJCE` class (4.2.6.5) to do the actual work.

Each credential generated contains a PKCS #12 keystore which is protected with a password. The password used is in fact the voter ID. This is somewhat curious, because it means that anyone could unlock the certificate knowing only the voter ID.

It appears that the security of this construction has been analysed and that the behaviour is correct (and secure in a protocol context), but when reviewing the code it looks rather strange, particularly because the source code does not comment on it.

4.5.2.9 SharesGenerationCommand

The class is used to generate El Gamal keys for the VCS and RCG, and this is done using the key generation interface exposed by the `KeyManager` class. Private keys are then split into shares, using the `CreatePrivateKeyShareWriter` and `ElGamalPrivateKeyShareWriter` classes from the same package, which again interface with the Shamir Secret Sharing implementation in jbasis.

Share files are signed and encrypted with a supplied RSA certificate. The storage medium can either be a smart card or a file, presumably the smart card option is what is being used for live keys.

4.5.2.10 EBPublicKeyCommand

The `EBPublicKeyCommand` generates the "public" El Gamal keys of the election for the Electoral Board, which satisfy the identity

$$pubKey_{EB} = pubKey_{VCS} * pubKey_{RCG}$$

Since the keys are public, this is completely straight-forward: load the public key arrays from the VCS and RCG, load the El Gamal group parameters for the election, and do pointwise multiplication of each public key pair.

4.5.2.11 ABWriteSharesCommand

The class is used to generate RSA keys for the Administration Board, split them into shares, and write the shares to smart cards. As such, it appears like a blend of several of the previous commands, doing the following steps:

1. Load the election certificate authority from a file, and enter the password to unlock
2. Generate a RSA key pair using the `KeyManager` interface. The key length is set to `Utils.RSA_KEY_SIZE`, i.e. 2048 bits.
3. Create a private key share writer, to split the keys into shares and persist them on smart cards. As in 4.5.2.9, there is also an option to write to file; this should only be used for testing.
4. Create a public certificate with the appropriate extensions, and use the CA to sign it.
5. Write the public certificate to a file.

This appears to be secure.

5 Conclusions and final recommendations

This review has touched on more than 80 classes implementing and handling cryptographic primitives, key generation, and other security functionality in the Internet voting implementation. In addition, a large number of auxiliary classes have been examined, albeit in less depth.

Our general recommendation is that key stakeholders review the technical chapters 3 and 4 of the report, and take appropriate actions to evaluate the findings presented, and assess and manage the associated risk(s). A short list summarising specific observations is also attached as Appendix A.

The reviewer's lack of prior experience with the electronic voting project has in some sense been a strength, by making it possible to analyze the implementation with fresh eyes; yet it is also a weakness, not least in the sense that it is hard to correctly judge the full context and thus the potential impact of each potential weakness found.

That said, based on the review that has been conducted, and our current understanding of the system, mnemonic would like to offer four main recommendations:

1. Address the security issue(s) caused by cryptographic weaknesses in the password-based encryption function `cipherSymmetrically` in the `UtilsRBAC` class (4.3.2.1).
2. Ensure that sufficient safeguards are in place to maintain the integrity of the audit events, despite potential vectors for dishonest parties to truncate or manipulate data from the secure-logger appenders. (4.4.1).
3. Verify that observations made relating to key generation (4.5) do not threaten the overall security of the protocol, by inadvertently revealing keys or other security-critical information.
4. Take concrete actions to improve the overall state of the source code, e.g. by consolidating and refactoring duplicate code, cleaning up interfaces and dependencies, documenting technical architecture and usage, and continually improving quality assurance practices in the development process.

It is our sincere hope that these points, and the review as a whole, shall be of use to the team attempting to test, improve, and extend the current Internet voting platform, and that it may also prove useful to other parties who, acting as academics or private citizens, want to inspect, study, validate, and improve the system.

To conclude on a positive note, mnemonic has not identified any critical cryptographic weaknesses that would make the system obviously unsuitable for use. Thus it remains only to wish the Ministry of Local Government and Regional Development, Scytl, and the e-voting project team the very best of luck with the upcoming trial in the 2013 elections.

A Appendix: List of findings

The below table summarizes the main (potential) security findings and observations on the Internet voting source code noted in the report.

Class	Chapter	Type of finding	Initial assessment
Third-party libraries	2.3	Old versions of third party security libraries	Consider impact of known vulnerability in Spring Security core
General	3.5	Use of SecureRandom with default RNG and provider	Consider making initialisation explicit
CryptoFactory	4.1.1.1	Defaults for symmetric encryption include 3-DES, ECB mode and weak password-based encryption	Avoid relying on jbasis default algorithms for symmetric encryption
BCPKCS7Envelope	4.1.2.4	Tries to generate 128-bit key for use with 3-DES	Key must be 192 bits
BigNumber	4.1.3.1	SecureRandom instance passed as argument to prime generation methods	Relies on caller initializing a secure random instance properly
	4.1.3.1	Optimized modular exponentiation using CRT may have timing side channel	Should only be used in situations where an attacker cannot obtain timing information
ShamirShadowManager	4.1.4.2	Shamir polynomial coefficients are selected from wrong statistical distribution	The bug invalidates the security proof for the secret sharing algorithm.
FileConnection	4.1.4.3	Method for persisting shares to files use weak password-based encryption	For security, shares should be persisted to smart cards using TokenConnection
TokenConnection	4.1.4.1	Method for persisting shares to smart cards use 3-DES	May be inconsistent with security requirements
AsymmetricCipher	4.2.1.1	Logic for handling	For data not fitting in a

		multiple-block data appears confused	single block, it would be more suitable to use a hybrid scheme (e.g. JCEEnvelope)
JCEEnvelope	4.2.1.3	The IV is encrypted, this is not necessary	No security significance, but somewhat unusual
	4.2.1.3	A message authentication code should be used on the enveloped data	Lack of MAC means that enveloped data will not satisfy strongest security notion (IND-CCA2)
	4.2.1.3	Note that the jbasis classes are used here, i.e. not the AsymmetricCipher from the local package	Ensure a single implementation is used consistently throughout
SymmetricCipher	4.2.1.5	Logic for handling multiple-block data appears confused	If instantiated with a non-block cipher (e.g. RC4), IV does not appear to be handled correctly
BlockElGamalEngine	4.2.3.1	No chaining of El Gamal blocks means that blocks may be shuffled or otherwise manipulated	Engine should not be used for arbitrary data, or data integrity must be protected elsewhere (i.e. by a signature or a MAC)
RCGCrypto	4.2.6.2	Insecure random instance used to select verification of zero-knowledge proofs	Evaluate impact of insecure random generation for security proof of protocol
VCSCrypto	4.2.6.3	The "…WithoutSigning" method variant may not be secure	Ensure that signing is used for production code
HardcodedElection-ManagementService	4.2.6.4	The hardcoded EMS contains a lot of fixed parameters, including credentials	Ensure that all "test" methods are properly overridden by the production EMS
CredentialsGeneratorJCE	4.2.6.5	Note that the common	This may be working as

		names of the voter certificates are not anonymous; anyone can compute the hash	designed; consult protocol specification
	4.2.6.5	Note overlap between CredentialsGenerator, CredentialsgeneratorJCE	Unclear why two classes is needed
HashedReturnCode	4.2.8.2	Truncated hash has only 160 bits of output	May be vulnerable to collision attack if adversary has partial control of return code inputs
com.scytl.evote.protocol.signers package	4.2.9	Note extensive code duplication between different Schnorr signature variants	All variants appear secure, but not obviously so; consider separating the Schnorr primitive from the protocol logic
		Signers pick the randomness from a different range than the encryption package	Either way is safe, as the probability of a "bad" choice is negligible
GeneratorSelector	4.2.10.2	The _paramSize attribute is only used for naming, so actual parameter sizes may not be consistent	Consider verifying that the length of the data corresponds with the parameter
CryptographicConstants	4.3.1.1	Specifies 3-DES as default symmetric cipher	May not be consistent with project requirements
CryptographicAlgorithms	4.3.1.2	generateSymmetricKey method builds a 192-bit 3-DES key	Keys generated are not suitable for use with the HMAC algorithm in the same package
UtilsRBAC	4.3.2.1	cipherSymmetrically method contains cryptographic flaws related to IV and password hashing	Method should be rewritten or replaced, as it has significant security weaknesses

TemporalDigestedFile	4.3.3.1	Use of MD5 for file integrity	May not be secure if an adversary can control inputs; replace MD5 with SHA256 or a MAC
com.scytl.slogger package	4.4.1	Conversions between String and byte[] are not locale aware	Inconsistent behaviour if locale can be modified
SignedEncryptedHMac-EventProcessor	4.4.1.7	Chained HMAC event log can be truncated back past a block boundary, at which point new events can be faked	Use a write-only medium and/or live monitoring as an additional safeguard against tampering
SignatureEventValidator / HMAC validators	4.4.1.8 / 4.4.1.9	Signature validator will accept events that do not contain a signature	Tricky interaction between the validators, seems fragile
AuditSecureFileAppender	4.4.2.1	Implements its own signer and decryption routines	Duplicates other project crypto classes, no clear rationale for duplication
RemoteConnector	4.4.2.2	Opens an insecure remote socket	Does not appear to be in use
Utils	4.4.5.1	Includes a fixed SHARE_PIN, "11111111"	Check whether this value is used by production code
InteractiveKms	4.5.2.1	Calls to cipher method use insecure encryption routine from UtilsRBAC	Fix and/or replace the code in UtilsRBAC
CAGenerationCommand, RSAGenerationCommand	4.5.2.2, 4.5.2.3	Note that password to CA keystore is written to disk as a plain text file	Ensure that file can not be accessed by unauthorized actors, and is securely erased from disk
SymmKeyGenerationCommand	4.5.2.7	Writes symmetric keys to disk using insecure routine from UtilsRBAC	Fix and/or replace the code in UtilsRBAC