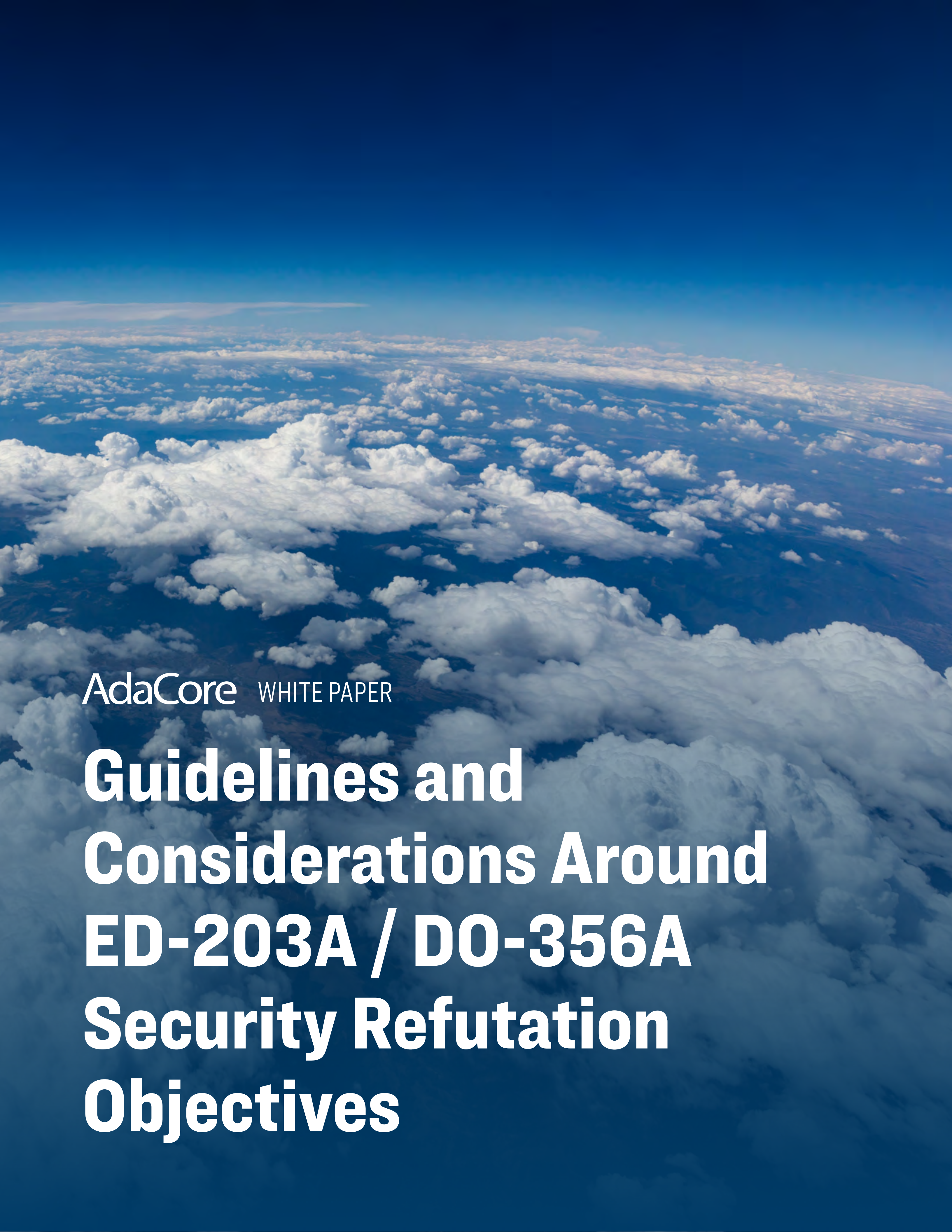

AdaCore WHITE PAPER

Guidelines and Considerations Around ED-203A / DO-356A Security Refutation Objectives

AdaCore

Guidelines and Considerations Around ED-203A / DO-356A Security Refutation Objectives

Paul Butcher

September 14, 2021

Contents

1 Introduction

This document has been produced as part of the “High-Integrity, Complex, Large, Software and Electronic Systems” (HICLASS) project. HICLASS was created to enable the delivery of the most complex, software-intensive, safe and cyber-secure systems in the world. HICLASS is a strategic initiative to drive new technologies and best-practice throughout the UK aerospace supply chain, enabling the UK to affordably develop systems for the growing aircraft and avionics market expected over the coming decades. The HICLASS project is supported by the Aerospace Technology Institute (ATI) Programme, a joint Government and industry investment to maintain and grow the UK’s competitive position in civil aerospace design and manufacture. The programme, delivered through a partnership between the ATI, Department for Business, Energy & Industrial Strategy (BEIS) and Innovate UK, addresses technology, capability and supply chain challenges. The £32m investment program, led by Rolls-Royce Control Systems, focuses on the UK civil aerospace sector but also has direct engagement with the Defence, Science and Technology Laboratory (DSTL).

HICLASS is divided into a number of work packages (WPs). WP1.2 was set up as a collaborative working group with the aim of analysing aerospace security standards, identifying maturity gaps within the standards and sharing best practice of security objective conformance. HICLASS document D1.2.4TH titled “Report on Identification of Aeronautical Security and Cyber-Security Processes” is the first deliverable of the “Security Process Standards Analysis” workstream and it was within this report that there was an identified need for additional guidance on Refutation Testing.

This report aims to offer further guidelines and considerations around security refutation objectives as described by the European Organisation for Civil Aviation Equipment (EUROCAE) ED-203A “Airworthiness Security Methods and Considerations” [6]. The ED-203A foreword states that ED-203A is technically identical to RTCA DO-356A [4] and this is also true of ED-202A [3] and RTCA DO-326A [5]. Within this report references will cite the ED-202A process specification and ED-203A methods and considerations, however, all guidelines and considerations offered within this report should be considered equally applicable for DO-326A and DO-356A.

Refutation is described within ED-203A / DO-356A as follows:

“Refutation acts as an independent set of assurance activities beyond analysis and requirements. As an alternative to exhaustive testing, refutation can be used to provide evidence that an unwanted behavior has been precluded to an acceptable level of confidence. NOTE: Refutation is also known as Security Evaluation in some contexts.”

The aim of Refutation in the context of the Airworthiness Security Process is to refute the allegation of exploitable vulnerabilities. The negative take on the wording within the phrase, as opposed to just claiming the system is secure, is deliberate. This is to emphasise the negative testing aspects of the associated activities and to differentiate the refutation testing phase from security verification testing. The difference is subtle but important to understand. Security verification activities (positive testing) focus on arguing, or proving where applicable, that a security requirement has been satisfied. Negative testing activities focus on trying to refute the hypothesis that the security requirement has been satisfied, whilst also gaining assurance that “unwanted behaviour has been precluded to an acceptable level of confidence”. Refutation differs to standard security verification testing in a number of ways. In some activities, like fuzz testing and penetration testing, the focus is on the observation of the system behaviour when the system is subjected to ’unusual’ scenarios, particularly where the expected behaviour has not been captured by requirements. The aim of the refutation activities is to identify any unexpected situations where the system would unexpectedly transition into an non-secure state (or more generally violate a minimal invariant guaranteeing the security of the system). Trying to identify these scenarios using formal standard requirements based testing methods would likely result in missed sequences; a better alternative is to target the testing on particular aspects of the system but then allow scenarios to be generated that are free from unwanted requirements based bias.

In addition, and in order to fully understanding the aim of Refutation, it is useful to understand what we mean by the term "Vulnerability" and also the difference and similarity between a "Software Bug" and a "Vulnerability".

ED-202A defines the term as follows:

“Vulnerability: A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (unintentionally triggered or intentionally exploited) and result in a security breach or a violation of the system’s security policy.” [3][5]

Here a software bug is only a vulnerability when it is has been demonstrated that it “could be exercised” (i.e. exploitable) and “result in a security breach or a violation of the system’s security policy”. However is also noted within ED-203A that:

“The situation that a vulnerability may be known as “bug” or defect for a long time before being recognized as a vulnerability should be considered. There are known cases where a vulnerability has been known for years as a defect without realizing the potential attacks. There are also cases where a vulnerability was considered “fixed” (by mitigation or prevention of known attacks), but shown by new attacks to still exist several years later.” [6]

This contradicts the ED-202A definition as the implication is that we should now also consider all known software bugs, without known attack vectors and where there was no known path to a resulting security breach or a violation of the system's security policy, as vulnerabilities (or at least "potential vulnerabilities").

This warning clearly identifies a potential safety risk in the classification of software bugs and/or the mitigation of identified vulnerabilities. This emphasises the importance in ensuring that the first phase of security vulnerability identification exercises the system in such a way that the maximum number of defects is identified. Phase 2 will then classify the defects as vulnerabilities (with calculated grades based on associated safety hazards) or non-exploitable software bugs (where the risk that this classification may change in the future is acceptably low). It is also vital that the arguments for classifying a software bug as a vulnerability (i.e. exploitable) are equally as strong as the argument for classifying a software bug as being non-exploitable.

Another consideration here is that ED-202A states that a software bug is classified as a vulnerability if can be "unintentionally triggered". This highlights another common issue with non-exploitable software bugs in that a change in the software could unknowingly transition the bug from being non-exploitable to a vulnerability.

Different refutation testing capabilities, and sometimes different strategies taken by a testing team, will determine if the activity itself is capable of classifying if the identified defect is exploitable or not. Noting that determining if an identified defect is exploitable does not automatically make the defect a vulnerability. As stated within the definition in order to be classified as a vulnerability the defect needs to be"(unintentionally triggered or intentionally exploited) *and result in a security breach or a violation of the system's security policy*". Activities such as penetration testing are ordinarily targeting entries to the system that are known to be capable of forming an attack vector. Here any identified anomalies will highly likely be exploitable. In comparison tools such as formal proof static analysers may identify scenarios that expose a defect. However, further analysis could determine that there is no known vector to allow an attacker to exploit the software bug. For example the analyser could identify a potential overflow exception due to the incorrect constraining of data types, however, it is later determined that the data required to trigger the constraint error is impossible to generate due to an input sanitiser at the system boundary. Fuzzing has the potential to identify both non-exploitable defects and exploitable defects dependent on the scope of the fuzzing campaign. If the fuzz test harness can be configured such that the fuzzing campaign targets an entry point to the system (where the entry is accessible to an attacker) then any identified anomalies should be considered as exploitable. If the campaign is scoped such that a number of tests are required to exercise individual parts of the system then any identified anomalies may or may not be exploitable.

The importance in determining if a software bug is actually a vulnerability will depend on the phase in the software life-cycle the system is currently in. It is widely accepted that software bugs take more effort to fix the later in the life-cycle they are identified and where applicable the best solution will always be to fix the software bug and re-test. If the application is deployed and in service then it may be infeasible to fix an identified and non-exploitable software bug. In this scenario a strong argument will be required showing convincing evidence that the known defect is not a vulnerability and cannot be exploited such that the software bug cannot lead to an aircraft safety hazard. The problem is that it may be difficult to argue that all bugs have been identified and, where applicable classified as vulnerabilities, if the Refutation activities are only focusing on known attack vectors. Either way if all defects and anomalies identified during Refutation are corrected or sufficiently mitigated it logically follows that there is a strong argument that the software system doesn’t contain any vulnerabilities.

In the context of the Airworthiness Security Process the overarching goal of the Refutation phase within the software development life-cycle is to provide assurance that the system is secure (and therefore safe). The goal of the refutation testing activities is to identify vulnerabilities within the system and to test the robustness of any implemented security measures.

In order to argue confidence over a claim that the system is secure all identified vulnerabilities need to be mitigated (or accepted) in one of the following ways:

- Fixed within the implementation (assuming the vulnerability is associated with a software bug).
- Logged within the vulnerability dossier and protected by a security measure, or a range of security measures, where the measures are commensurate with the safety hazard associated with the exploitation of the vulnerability.
- Logged within the vulnerability dossier and mitigated through a mechanism at a higher level in the overall aircraft platform.
- Logged within the vulnerability dossier and accepted that the calculated risk of the vulnerability being exploited is sufficiently low.

If no vulnerabilities can be identified then one of the following scenarios has occurred:

- Refutation could not identify any bugs or anomalies.

- Bugs or anomalies have been identified, however we can sufficiently argue that the risk of the identified bugs and anomalies being exploited is acceptable.
- Bugs or anomalies have been identified and one or more of the identified bugs or anomalies has been demonstrated to be exploitable (unintentional or intentional), however we can sufficiently argue that the risk of the exploited bugs and anomalies resulting in a security breach or a violation of the system's security policy is acceptable.

In additional a poorly planned or executed Refutation phase may of course fail to identify all of the bugs or anomalies. In this scenario vulnerabilities may exist in the system, however they are yet to be discovered.

ED-203A / DO-356A states that refutation encompasses the following testing and analysis activities:

- Security penetration testing
- Fuzzing
- Static code analysis
- Dynamic code analysis
- Formal proofs

Full definitions of these activities can be found within Annex sections (as and when the correlating Annexes are made available), however, for the scope of the main report the following overview definitions should be used:

Security penetration testing: a testing mechanism involving the attempted exploitation of a system vulnerability with the aim of reading or altering the state of an identified security asset.

Fuzzing: an automated test-case input generating and system anomaly detection mechanism. Fuzzing capabilities usually involve the mutation of a starting corpus of test case inputs with the aim of generating more test cases that the system is then subjected to whilst any unusual behaviour is observed. The goal of fuzzing is to identify system inputs that can transition the system into an non-secure state.

Static Code Analysis: a semantic analysis at the source code level of a system, performed with the aim of identifying code constructs that may be considered unsafe and / or non-secure.

Dynamic Code Analysis: the analysis of the behaviour of the system whilst the system is executing. An example of dynamic code analysis would be the monitoring for a non-safe and / or non-secure sequence of instruction calls made to the processor (i.e. detection of buffer overflows). Dynamic code analysis can be enforced within the semantics of programming languages via run time constraint checks or via tools that detect memory corruption bugs via code instrumentation added during additional compilation passes.

Formal Proofs: an unambiguous and programmatically verifiable finite sequence of formulas (axioms, assumptions, rules and inferences - also known as contracts and assertions) that are constructed during software design (and automatically via interactive theorem proving tools) with the aim of describing the intended behaviour of the system such that the formulas can be statically checked via mathematical proof checkers. Where systems have been developed with a "Design by Contract" methodology, and formal methods are used to specify security requirements within the software implementation, static verification tools can be used to generate security assurance over the information flow through the system. In addition automated formal proof tools can be used to generate assurance over the absence of run-time errors (that could lead to denial of service attacks).

It should be recognised that this is not an exhaustive list of refutation activities and there may also be a benefit to overlap and / or combine the disciplines. For example some Fuzzing implementations will use Dynamic Code Analysis to detect non-secure system behaviour.

2 Purpose

To provide guidelines and planning considerations for the following issue identified within the HICLASS report 1.2.4TH titled “Report on Identification of Aeronautical Security and Cyber-Security Processes”:

Issue 15.

“ED-203A includes objectives and activities for “refutation testing”. However, deciding the right refutation/penetration testing to do (in terms of quantity and quality) is difficult in practice, and no guidance is given. For example, how much is enough? What is tested (e.g˙ model and/or auto- generated code)? How does the choice of programming language (e.g. C, Ada) affect it? Additional guidance would be beneficial here.”

It is, however, recognised that there will be circumstances when some considerations cannot have correlating concrete guidelines, for example when the consideration is broad reaching and/or project specific. In these scenarios the annex entries should try and provide further assurance activity specific considerations to allow the reader to conclude their own project specific refutation planning activities. It is also reasonable for a consideration to not be applicable to a particular activity. This scenario is considered acceptable when a valid explanation is provided. In addition, there may be additional activity considerations that are not included in this main report but are applicable to a Refutation activity and therefore will need to be captured within the appropriate annex.

3 Scope

This report focuses on how to plan for security refutation test activities and what needs to be considered in order to meet the Security Refutation Objectives defined within ED-203A / DO-356A. In addition, this report assumes the reader has a good understanding of the ED-202A / DO-326A process specification and the ED-203A / DO-356A methods and considerations. However, it is beneficial in describing the scope of this report to reiterate some of the fundamental terms and concepts within ED-202A / DO-326A and ED-203A/DO-356A. In addition, and particularly where an interpretation of the process has had to be made, assumptions over the relevant process stages will be described to provide clarification.

The security refutation objectives, as stated within ED-203A/DO-356A are as follows:

- *O3.1 Refutation analyses are performed to identify new vulnerabilities.* [6][4]
- *O3.2 Refutation tests are performed to evaluate the exposure of vulnerabilities in the security environment and to challenge the vulnerability evaluation.* [6][4]
- *O3.3 Refutation test plans are available. Refutation test results cover refutation test plans and performed tests. Refutation test results are analyzed and discrepancies are justified and traced.* [6][4]

Therefore the main objective of the security refutation is to identify vulnerabilities within the system. A secondary, but also mandatory, objective is to ensure each identified vulnerability can be rated for exploitability; i.e˙ can the software bug, hardware flaw etc form part of an attack path. The third objective can be broken down into three sub-objectives:

1. A refutation test plan has been produced as part of the Plan for Security Aspects of Certification (PSecAC).

Ordinarily PSecAC approval will be required from the certification authority. The exception to this rule is when input activities (i.e. the Aircraft Security Scope Definition (ASSD output), the Aircraft security architecture(ASAM output), and the Aircraft threat conditions) have indicated that the severity of the aircraft security effect is either “No Effect” or where a “Minor” effect has been negotiated away with certification authorities. Either way the PSecAC needs to show how the plan will address the regulatory requirements for security and this should include a detailed breakdown of the planned refutation activities.

2. Ensure that the refutation testing performed covers the activities described within the refutation test plan.

   As with all test related activities it is essential to show that the activity:

   - is fit for purpose (are we testing what we stated we would be testing),
   - has been performed correctly (has the test been executed within a sufficiently controlled environment such that there is confidence that the test execution has correctly assessed the targeted security aspect),
   - was recorded and is repeatable (has the result of the test been saved and can the test be re-executed such that the same behaviour and results are observed).

3. Ensure test results have been analysed and understood and where applicable false positives and false negatives have been documented.

   It is expected that many accepted forms of refutation testing may have known limitations to their vulnerability finding capabilities resulting in false negatives. In addition, tools like static analysers can often result in false positives resulting in user required interaction to overcome limitations in the analysing algorithms. Where this is the case it is essential that the limitations are understood and documented within the PSecAC and that a process is adopted that ensures the analysis of the test results takes the known limitations into account. In many cases conformance to a commensurable DO-330/ED-215 Tool Qualification Level (TQL) can help provide the information required to understand the limitations of the refutation testing tool. However, ED-203A states the following:

- 4.2.8 “Tool Security Objectives”
  - “Tool that is used to detect vulnerabilities
  - Such tools may be qualified according to ED-215 [8] / DO-330 [1] TQL-5 .
  - Relevant tools are the ones that can fail to detect vulnerabilities on the product under development, such as static code analysis. Such tools may be qualified according to ED-215 / DO-330 or ED-80 / DO-254.
  - Such tools, scripts and supporting data that are only used within refutation testing to execute attacks are not subject to security tool qualification.
  - NOTE: TQL-5 qualification objectives may be satisfied without any tool qualification data from the tool developer, as described in ED-215 / DO-330 section 11.3.” [6][4]

Note that regardless of if the refutation testing tool assurance can be obtained via satisfaction of relevant TQL objectives or not the known limitations of the tool should be documented within the PSecAC.

In addition, security refutation activities should adopt a layered approach where the limitations of one activity are mitigated by the capability of another. This line of thinking also follows the “Defense-in-Depth" philosophy that ED-203A proposes as a consideration of the development of the security architecture and measures. Here is is recognised that:

> “Defense-in-depth is important as multiple lines of defense are the preferred means to defend against multiple threats of varying complexity... As attacks can be of numerous types, and to be prepared for unknown attack techniques, different technological concepts are to be used in different layers when defending against threats.” [6][4].

The same is true when attempting to identify vulnerabilities; it is unlikely that one refutation testing methodology will be sufficient on its own. An example of complementary security refutation activities would be to use fuzz testing to gain confidence in the absence of run time errors of a sub-system of an application that cannot easily be achieved through formal proofs. The security refutation activities adopted and how they complement one another, should be described within the security refutation test plan.

It is also important, when planning for security refutation, to understand the difference between activities that focus on known attacks and the activities that look for potential vulnerabilities that may or may not form part of an attack path.

Both techniques are equally applicable and if both are adopted, greater security assurance can be achieved.

ED-202A / DO-326A and associated documents don't explicitly mention or define the term "Security Refutation". This is explained in ED-203A / DO-356A as follows:

> "ED-203A / DO-356A introduces the term "Refutation" to describe and collect assurance activities as a method for airworthiness security. This term is new because current versions of ED-202A / DO-326A do not use the term, nor do other published aviation industry documents (ED-79A / ARP4754A, ED-12C / DO-178C, ED-80 / DO-254, etc.). Even though these standards do not use the term "refutation", these standards do discuss many refutation activities as part of activities associated with safety/security assurance objectives." [6][4].

Refutation and Vulnerability testing are therefore new and, prior to ED-202A and ED-203A, not considered standard practice within airworthiness certification. As such the discipline is lacking well established methods, considerations and guidelines and this report will attempt to address some of these issues.

Security refutation clearly plays a key role in security assurance as it forms part of the vulnerability identification and security measure evaluation activities within the Airworthiness Security Process. In the context of ED-202A / DO-326A security assurance is described as:

> "...necessary to assure that security measures perform as intended and that the final product is acceptably free of known and exploitable vulnerabilities, which may be introduced during development." [3][5].

However, it is still not straightforward to correlate the activity into the process specification which can lead to confusion when trying to determine where to reference a Refutation plan within the Plan for Security Aspects of Certification (PSecAC). The ED-203A / DO-356A description does state that: "Refutation is also known as Security Evaluation" which implies that within ED-202A / DO-326A Refutation falls under the remit of "Security Verification"; a described purpose of security verification is to contribute to the evaluation of security effectiveness.

> "Security verification has the following purposes:
>
> ...
>
> - Contributes to evaluation of security effectiveness (specific to security)."[ED-202A / DO-326A] [3][5].

Within the Airworthiness Security Process the Security Effectiveness requirements are derived during development of the “Aircraft Security Architecture and Measures" and assessed against the “Preliminary Aircraft Security Risk Assessment". The Security Effectiveness requirements form an input into the Aircraft Security Verification (ASV) and a goal of this exercise is to:

> “Finalize vulnerability test plan and evaluate how well it covers the security requirements in its context of the threat scenarios in the security risk assessment, including its security effectiveness." [ED-203A / DO-356A] [3][5].

For the most part Security Requirements Verification can be achieved through conformance with standard verification techniques, as described within DO-178C [2]. This will, however, be dependent on the individual form the security requirements take, the assumption here is that standard verification techniques can be adopted because the requirements are expressed in a positive (finite test cases) notation. When this is the case is not expected that security refutation testing will be required to add to the verification of security requirements. If the security requirements are expressed in a notation that favours negative testing then it may be feasible to utilise refutation testing activities to provide evidence that the requirement is satisfied, however, how this is achieved is beyond the scope of this report.

In addition, another goal of the ASV is to:

> “Conduct vulnerability assessment and testing and analyze results for security risk." [ED-203A / DO-356A].

This objective can be achieved by the adoption of security refutation testing methodologies. In addition, the results of the security refutation testing will form part of the Aircraft Vulnerability Dossier. The aircraft security verification and test results and analysis, achieved via the ASV, are also required to be documented within the PSecAC summary report.

ED-202A / DO-326A states that the purpose of Security Verification is:

- Verification that systems, hardware, and software meet security requirements (normal purpose in a standard development process such as ED-79A / ARP 4754A), and
- Contributes to evaluation of security effectiveness (specific to security),
- Identification of vulnerabilities for final Security Risk Assessment (specific to security).[ED-202A / DO-326A].

The AWSP then expands on this by describing 3 testing activities:

- Security Requirements Testing.
- Security Robustness Testing.
- Vulnerability Testing.

ED-202A / DO-326A states that Security Requirements Testing and Security Robustness Testing:

> “demonstrate that the implementation of the technical security requirements is correct even under abnormal inputs and conditions.” [ED-202A / DO-326A].

The first two test activities are therefore based around the verification of security requirements and, as previously discussed, can be considered “standard practice", particularly when developing airborne software that has been associated with safety related failure conditions. Albeit that the focus of these tests will be on security requirements rather than safety requirements.

The third set “Vulnerability Testing" doesn’t focus explicitly on requirements and can be met through the adoption of applicable security refutation activities.

Airworthiness security process specifications extend existing software considerations in airborne systems and equipment certification by introducing the concept of a “Threat Condition". “Threat Conditions" differ from "Failure Conditions" in that they introduce the concept of:

> “... acts of intentional unauthorised electronic interaction (IUEI), involving cyber threats..." [ED-202A].

Vulnerabilities are described within ED-203A as:

> “A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (unintentionally triggered or intentionally exploited) and result in a security breach or a violation of the system’s security policy." [ED-203A].

Threat Conditions therefore include the exploitation of system vulnerabilities, intentionally or unintentionally triggered, where the exploitation can have a direct, or consequential effect on the airplane and/or its occupants. The intention of the assurance activities, covered under Refutation, is to argue that an acceptable level of confidence can be demonstrated in the robustness of the system.

In this context a "Robust System" is one that does not contain vulnerabilities that could lead to Threat Conditions.

In order to understand Refutation activities, in the context of an Airworthiness Security Processes (AWSP), and the critical role it plays in achieving certification, it is important to consider the primary goals of the AWSP. ED-202A / DO-326A describes the goals within Chapter 2 as:

> “The purpose of the Airworthiness Security Process (AWSP) is to establish that, when subjected to unauthorized interaction, the aircraft will remain in a condition for safe operation (using the regulatory airworthiness criteria). To accomplish this purpose, the Airworthiness Security Process:
>
> - Establishes that the security risk to the aircraft and its systems are acceptable per the criteria established by the AWSP, and
> - Establishes that the Airworthiness Security Risk Assessment is complete and correct.” [ED-202A / DO-356A].

Refutation testing plays a role in gathering evidence to support an argument that “the security risk to the aircraft and its systems are acceptable". In addition, only with a suitable refutation security test plan can the regulatory authority establish that the “Airworthiness Security Risk Assessment is complete and correct".

The number of security testing disciplines that fall under the realm of “Refutation Testing” and make for potential methods for achieving compliance with ED-203A / DO-356A Security Refutation Objectives are vast. In addition, new and existing testing capabilities and testing methodologies are rapidly evolving as existing and new software vulnerabilities are understood.

ED-203A / DO-356A states that refutation encompasses the following testing and analysis activities:

- Security penetration testing
- Fuzzing
- Static code analysis
- Dynamic code analysis
- Formal proofs

However, this is not an exhaustive list and other existing and future activities may also play key roles. In addition, ED-203A was established to provide guidelines for both ED-202A and RTCA's equivalent: DO-326A and it is recognised that alternative methods may be available to also obtain certification.

The main section of this report will identify a number of refutation test considerations. Annex parts will then be provided for each of the activities listed. This allows the report to be expandable with new refutation activity annexes added as and when they become available. Each annex will address the considerations by stating their relevance to the activity whilst also providing guidelines for meeting the security refutation objectives where applicable.

The guidelines within this report will try not to refer to particular commercially available tools, to avoid any bias. In addition, programming languages will only be considered when they can add value on addressing a security related consideration.

This report will propose considerations for Refutation Testing aspects of a Refutation Test Plan that fits within the "Plan for Security Aspects of Certification (PSecAC)" as defined within the ED-202A / DO-326A Airworthiness Security Process Framework.

Other activities including system functional testing, security measure testing and security robustness testing will not be covered.

4 Considerations

4.1 General Considerations

4.1.1 Perspective

Existing, and well established, airworthiness safety processes do not consider intentional disruption and instead focus on aspects like environmental hazards and failing internal components that could lead to accidents. This shift from safety "failure conditions" to security "threat conditions" requires a change of perspective and an understanding of the differentiation between verification and refutation. ED-203A / DO-356A emphasises the point when it states:

> "Verification and refutation activities should be performed separately because they follow different concepts. Verification activities are requirements-based while refutation activities need to be performed from an attacker perspective." [ED-203A / DO-356A].

Therefore a key aspect of refutation testing, which differs from standard verification based test approaches, is the perspective that the refutation testing team should adopt when determining a test strategy and test plan.

Whilst a standard verification testing approach should be adopted for arguing the satisfaction of security requirements, the likelihood of activity success, in terms of finding exploitable vulnerabilities, will be increased if the frame of reference is considered from an attacker's perspective. In reality this can be a very difficult approach to adopt and even harder to measure success. However, a key aspect of this transition is to ensure that, where applicable, the refutation activities are adopted that mirror known approaches taken by attackers to achieve their goal of unauthorised electronic interaction.

4.1.2 Security Coding Standards

ED-203A / DO-356A objective O2.1 states:

> "Vulnerabilities in security measures and assets (including COTS) are identified and evaluated for their potential impact on safety." [ED-203A / DO-356A].

In addition, a note has been tagged to the objective stating:

> "The situation that a vulnerability may be known as "bug" or defect for a long time before being recognized as a vulnerability should be considered." [ED-203A / DO-356A].

Software bugs therefore need to be identified as part of "Vulnerability Identification". If the vulnerability identification is being performed as part of the development of a new or an update to a system the software bug can and should be fixed. If a software patch is not an option or an established system is undergoing retrospective Vulnerability Identification then any identified bugs or anomalies will need to be assessed for "exploitability". This assessment requires the analysis of the bugs or anomalies to determine if an exploit can be used to form an attack across the defined system security perimeter. All identified vulnerabilities are required to be documented within the Vulnerability Dossier.

In addition, ED-203A / DO-356A implementation objective O8.2 states:

> "Source code and/or hardware description conforms to security coding standards." [ED-203A / DO-356A].

It is widely recognised that software bugs can be introduced during the software implementation phase through a lack of developer understanding of the security risks associated with particular programming language semantics. One way of mitigating these risks is through the adoption of a suitable security focused coding standard. The construction of a coding standard should be based on considerations of known vulnerabilities within the programming languages used to develop the airborne system. One method of constructing an argument that an adopted coding standard captures all known security vulnerabilities associated with the chosen programming languages is to ensure it conforms to the guidelines set out within the International Organisation for Standardization (ISO) Technical Report 24772 titled: “Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use".

This report recognises that:

> “All programming languages contain constructs that are incompletely specified, exhibit undefined behaviour, are implementation dependent, or are difficult to use correctly. The use of those constructs may therefore give rise to vulnerabilities, as a result of which, software programs can execute differently than intended by the writer. In some cases, these vulnerabilities can compromise the safety of a system or be exploited by attackers to compromise the security or privacy of a system."

In addition, the report proposes that by identifying programming language constructs that could lead to vulnerabilities, and by providing alternative and secure approaches, multiple common vulnerabilities can be avoided altogether.

4.1.3 Quantity and Quality

Refutation test considerations can be categorized based on the following two metrics:

- Quantity - how much is enough?
- Quality - what constitutes ‘strong refutation evidence’?

Quantity is a difficult problem as the identified refutation activities often adhere to a ‘negative testing’ philosophy where the number of the required test-case permutations makes exhaustive testing infeasible. Functional requirements stipulate how the system should behave and can be verified by ‘positive’ testing. Negative testing focuses on the inverse of the required behaviour by attempting to address the “what if?". However, with some requirements the inverse scenarios could be limitless and human preconceptions often rule out scenarios that could lead to the exposure of a vulnerability. In addition, traditional coverage criteria objectives, as found within safety standards like DO-178C, were not designed with security objectives in mind. Scenarios including retained state and dynamic memory allocation are not always sufficiently considered when measuring coverage and therefore cannot be relied upon to make any security arguments. This is better described by the following scenario:

> A set of test-cases are created that target a software unit. By analysing the control flow of each test execution, coverage metrics are calculated and 100% Modified Condition Decision Coverage (MC/DC) is recorded and no bugs are detected. The application is then tested again with the same test-cases but this time the system state is retained in-between test executions such that there is an incremental execution that can now reach different application stages. 100% MC/DC is again recorded, however, this time a buffer overflow fault is identified.

The problem is exacerbated if the bug was only identifiable when the test suite was executed multiple times, perhaps more than would be considered necessary by a test engineer. Therefore 'quantity' considerations should include a coverage criteria suitable for security based testing. In addition, it should be acknowledged that achieving 100% of any coverage criteria does not guarantee a system is secure or safe.

Quality considerations will focus on the level of security assurance that each individual refutation activity can provide. The quality of refutation test evidence, for each identified activity, should be measurable and stated within the test plan. Each activity will need to consider what evidence it can bring to support the argument and how it is commensurate with the identified Security Assurance Level (SAL) of the associated Security Measure the activity is testing.

The final consideration asks each refutation test activity to consider how it fits into the overall goal of the refutation plan and to which of the other activities it is related. It is not expected that a single refutation activity will be sufficient to form a suitable refutation test plan. Instead a combination of multiple activities will likely be required to ensure suitable confidence can be gained to form an argument that the system is free of unwanted behaviour. This consideration requires a clear understanding of the strengths and weaknesses of each activity and the software and hardware tools used to achieve the goals of the activity. An example would be using static analysis to identify a coding vulnerability, for example, SQL injection through unsanitised password entry fields, and then using fuzz testing to identify a software bug, where the attack vector is the same password entry field, and where the exploited bug could result in a denial of service attack. In this example two refutation activities are used to identify two different exploitable vulnerabilities, however, it may not be the case that any one activity could identify both.

4.2 Activity Specific Considerations

Each consideration below should have a corresponding section within each Refutation Activity Annex which should describe if and how the consideration is relevant to the particular activity.

For the majority of the guidelines and considerations, unless not applicable for the particular activity under focus, it is beneficial to the reader to show examples of how the action resulting from the consideration could be described within the Refutation Test Plan.

1. **Verification and Refutation Separation**

   ED-203A / DO-356A stipulates that Verification and Refutation should be treated as separate activities and whilst this recommendation isn't a direct objective there is a benefit to understand and document how the refutation activity conforms to this guideline.

   1.1 What evidence can be collected to argue that a level of separation exists between verification and the refutation activity?

   For example, how is the activity performing non-functional testing that is not directly focused on the verification of requirements (including security related requirements).

   1.2 What argument can be made over how the process for performing the refutation activity ensures the testing is also performed from the perspective of an attacker?

   The aim here is to try and ensure the activity focuses on both vulnerability identification and defensive security measure assessment.

   1.3 How does the activity deal with human preconceptions and unwanted bias within test cases?

   This is an opportunity to promote any automated negative testing aspects of the activity and to identify manual aspects that could introduce an unwanted bias. An example here is automated test case input generation from mutation algorithms adopted by fuzz testing. In this example the automation of the test input creation demonstrates independence from manual input creation which can add an unwanted bias through the knowledge of the application design.

2. **Assurance Goals**

Refutation testing is used to aid security assurance arguments, however, it is expected that particular activities will target different aspects of security assurance. For example, formal proof could be used to argue that a system, or part of a system, is absent of run-time errors and therefore free of vulnerabilities. Fuzz testing may be using specific algorithms to target common vulnerabilities like buffer overflow attacks.

This is broken down further into the three sub-considerations that should be answered and documented within the Refutation Plan within the PSecAC.

2.1 What are the main goals for the refutation activity? What assurances is it trying to achieve and how?

2.2 What unwanted behaviour is the activity trying to exercise in the system?

2.3 Is the activity targeting a known set of vulnerabilities/exploits or is the activity attempting to capture any possible vulnerabilities through techniques like brute force test case injection?

3. **Limitations**

The limitations of the activity need to be clearly understood and documented.

3.1 What are the known limitations of the refutation activity?

For example if fuzz testing is being used to identify buffer overflows is the test application runtime guaranteed to always detect a buffer overflow? This may be the case for applications developed in Ada but not always the case for applications developed in C.

3.2 What are the known factors that can stop the activity from completing its goals?

For example, if formal proof is being used to argue the absence of runtime errors can the adopted formal proof static analyser discharge all verification conditions required to guarantee that all scenarios have been considered?

3.3 Are there any known hardware or system design architecture considerations that could affect the capability of the refutation activity (i.e. choice of Instruction Set Architecture, programming language or compiler optimisation).

An example is where runtime constraint checking is enforced to ensure all software bugs can be identified during fuzz testing, however this is not representative of the final target build where the runtime checking is not included. In this scenario any performance implications of the constraint checking need to be considered.

4. **System Complexity**

It is expected that system complexity has a direct correlation to the likelihood of the activity identifying the maximum number of vulnerabilities within the system. The same is true for the likelihood of being able to suitably measure the assurance of a targeted defensive measure. It is important to understand how system complexity is being measured in order to be able to assess the suitability and capability of the particular refutation activity.

In this regard system complexity is relevant to the particular activity and it is understood that complexity measurements for one activity will differ to another. For example, Penetration Testing will likely be interested in the availability of public library versions embedded within the codebase such that it can identify known exploits for particular software versions. Fuzz testing may see code bases that use comparison statements with complex types as complicated. Activities involving theorem provers will want to measure state explosion and loop termination.

Once a complexity measurement has been described it is important to understand at what point does the measured complexity start to affect the capability of the activity.

Once a complexity measurement has been described it is important to understand at what point does the measured complexity start to affect the capability of the activity.

4.1 What is the definition of "system complexity" for the particular refutation activity?

4.2 How does an increase in complexity affect the scalability of the activity?

5. **Measuring Confidence**

   It will be key to security critical assurance refutation activities that the assessor can understand what evidence the activity can produce in order to measure the security assurance of the system.

   5.1 What evidence can the refutation assurance activity provide to measure that unwanted behavior has been precluded to an acceptable level of confidence?

   For example, can a Fuzz testing capability show the areas of the control flow that have, and have not, been exercised. If Formal Proof has been adopted can the activity produce clear evidence of the assurance that a system, or part of a system, is absent of runtime errors?

   5.2 What format will the activity use to record newly identified vulnerabilities?

   5.3 How can the results of the assurance activity be reproduced and what information needs to be configured to achieve this?

   5.4 What coverage criteria will the activity use and what are the known limitations for each case?

   Example coverage criteria algorithm limitations are discussed in Section 4.1.3 "Quantity and Quality".

6. **Scope**

   This area of consideration focuses on how the scope of the tested area of the system is documented and what particular aspects of the system the activity can, and cannot, test.

   6.1 At what scope will the activity be performed (i.e. system, subsystem, item or other)?

   When collating security assurance evidence for an existing platform it is expected that Refutation testing will be, or may have to be, performed at the system level. When the Airworthiness Security Process is adopted from the start of a new development it may be possible, and in many cases beneficial, to reduce the scope of the tested aspects of *the system through sub-system or unit based refutation testing. If the activity is not testing the entire system how is the scope of the test being documented?*

6.2 What evidence can be provided to show that the activity has sufficiently tested the security measure/s commensurate to the Security Assurance Level associated with the measure/s?

When Refutation Testing is being used to provide assurance that a security measure, or set of security measures, are effective in their role of guarding a security asset it is important to present a strong argument that the Refutation Activity is fit for purpose. For example, Penetration Testing that uses an outdated set of known vulnerabilities may not be considered may not be considered appropriate on it's own to provide the required assurance of a Security Assurance Level 4 security measure.

6.3 What software representation level is the activity aimed at (i.e. source code, executable code, or some other intermediate representation)?

Static analysis techniques for identifying vulnerabilities tend to be aimed at source code. Fuzz testing is performed on the executable code. Other refutation testing activities like symbolic execution may be aimed at an intermediate representation that is executed through an emulated environment.

6.4 What are the limitations of targeting the activity on a particular software representation level, and how does this affect the system behaviour?

For example, if the refutation testing is performed on an intermediate representation of the system source code or the actual source code and not the binary executable image, what assurance can be provided that additional vulnerabilities will not be created during the compilation phases required to create the binary executable.

6.5 Is the hardware test-harness representative of the final implementation, and if not how does this affect the system behaviour?

For example, where an emulator is being utilised on a host environment to represent the system under test, proof is needed that the emulator is a true representation of the target hardware.

6.6 Does the activity need to alter the software under test (i.e. insert instrumentation code for metric-related data collection) and how does the alteration affect system behaviour?

For example, if a fuzzer is instrumenting the binary code for control flow path execution tracking, then guarantees are required to ensure that the additional instrumentation instructions do not inadvertently alter the original program behaviour.

6.7 Can the activity test the boot process and verify that integrity is protected?

Where applicable systems may require complete end-to-end security including providing assurance that the integrity of defensive security measures with the system boot process cannot be compromised. For example, if the system has implemented a drive encryption scheme or a firewall is the activity capable of attempting to bypass the security measures.

6.8 What aspects of the system can not be exercised by the activity?

For example it is unlikely that fuzzing would be suitable for testing the boot process integrity whilst limited guarantees may be made when applying formal proof to a system that encompasses various prebuilt low assurance software libraries.

7. **Threat Scenarios**

   As and when threat scenarios are identified, refutation activities should be run to target the implemented security measures.

   7.1 How can the activity be configured to challenge the vulnerability evaluation?

For example, can the activity demonstrate that vulnerabilities, as documented within the vulnerability dossier, can not be exploited.

7.2 What identified threat scenarios can't be covered by the activity?

It should be clear within the vulnerability evaluation which refutation activities are being employed to evaluate the exploitability of each vulnerability.

8. **Pedigree**

   This set of considerations focuses on the assurance aspects of the actual refutation testing activity and in particular the software and hardware tools required to execute the test. Whilst it is recognised that refutation testing tools do not need to be implemented to a particular DO330 Tool Qualification Level (TQL) it is also reasonable to assume that any known tool assurance is documented within the Refutation Testing plan.

   The following considerations try to breakdown an assessment of the assurance of a Refutation Testing tool, particularly where a TQL assessment cannot be provided.

   8.1 Has the activity been used for establishing security airworthiness before?

   8.2 Is this a mature approach of vulnerability identification or experimental?

   8.3 If the activity involves an external to the organisation entity, what evidence (certification packs etc) can the subcontractor provide?

   8.4 How much human interaction does the activity involve and how can you demonstrate that the operators have the required level of experience for the associated tools?

9. **Relationship to other refutation activities**

   It is expected that in many scenarios a single refutation testing activity will not be suitable on its own to provide suitable security assurance. In addition, many refutation testing activities will contain known limitations that can be satisfied by other refutation testing activities. An example is where formal proof is used to argue the absence of runtime errors within the implemented application code but may not be capable of providing assurance over third party software libraries. In this scenario fuzz testing may be also applied to provide further system security assurance.

> 9.1 What other activities complement this activity and which other activities can make up for the known limitations of this activity?

Annexes

A Fuzz Testing

A.1 Activity Description

Traditional Fuzz testing, also known as ‘fuzzing’, is a software testing technique that involves the automatic generation of system inputs generated through mutation of a user-provided set of test cases, also known as the starting corpus.

Fuzz testing focuses on robustness rather than verification; the goal of the exercise is to determine if the system can protect identified security assets when subjected to unusual operating conditions (i.e. test cases that include unconstrained data). This is achieved by a monitoring process that, during test execution, attempts to detect if the system has crashed or hung. With fuzzing, under some circumstances, for example the use of design by contract pre- and post-conditions, it is also feasible to detect a system transition into an unwanted or unknown state. However, it is not feasible to achieve guarantees that the implementation conforms to the functional specification with fuzzing; formal verification will need to be adopted to achieve this.

The term fuzz testing covers a wide range of automated testing capabilities. In particular fuzz testing methodologies tend to be either black box, grey box or white box. Black box fuzz testing is where the software system is tested without the need for any knowledge of the internal workings of the application (i.e. the source code). With black box fuzz testing the binary executable of the software system is usually executed through an emulated environment or a virtual machine. Grey box fuzz testing requires some instrumentation of the system under test and therefore the source code must be made available. The instrumentation is usually added via a compiler wrapper and no knowledge of the functional specification of the system is required (i.e˙ the source code must be made available to the compiler, however an understanding of the implementation within the source code is not required). Fuzz testing can also be used to identify bugs within sub-systems, items or units and a white box fuzz testing approach can be adopted to ensure the correct test scope is defined and specific subprogram signatures are targeted for test injection. White box fuzz testing can also be useful to identify areas within the code base that the tester specifically wants to fuzz (i.e˙ a security measure protecting a security asset). Manual (or automated if available) test-cases can be created and added to the starting corpus to ensure the required coverage is achieved.

It is expected that in most cases the attacker will only have access to a black box fuzzing capability, however, this should not be relied upon; particularly when code reverse engineering from a binary executable is a possibility. In addition we need to accept that publicly available software libraries may have been embedded into the system under test and that these libraries may contain vulnerabilities (known or unknown). Where a public library is used the attacker will have full access perform white box and grey box fuzz testing on the full API of the library and can use any found vulnerabilities to their advantage.

American Fuzzy Lop (AFL) [9] is one such fuzz test mutation engine that supports black box, grey box and white box fuzz testing.

A.2 General Considerations to achieving effective fuzz testing

Fuzzing is employed in cases where manual robustness testing and vulnerability identification mechanisms do not scale. This is often the case with large applications and applications with complex code semantics. In these cases, it is difficult to manually create test cases to sufficiently exercise the program. Furthermore, fuzzing removes the need for human driven manual test generation which could be bias and error prone.

For fuzz-based testing to achieve maximum effectiveness in vulnerability identification the test environment should enforce the maximum usage of available anomaly detectors. For example, programming languages such as Ada include run-time checking of multiple constraints (including memory overflow checks and developer embedded contract assertions) and these should be included within the fuzz test harness. For languages with limited runtime checks, such as C, third party anomaly detectors should be enforced (for example: invalid memory access detectors). Failing to enforce anomaly detection can lead to false negatives and an incorrect level of security assurance assumed.

A.3 Coverage and Fuzzing Campaign Criteria

Most fuzz testing capabilities have some understanding of the level of coverage of the system under test that has been exercised during a fuzz testing session. Coverage in the context of fuzz testing will be tool independent and will range from a measurement of instrumentation points added during test harness compilation, to a more advanced analysis of the source code coverage. It is noted within section 2.1.2 of the main report that there are limitations with all coverage algorithms and achieving maximum coverage based on rigorous algorithms like Modified Condition/Decision Coverage (MC/DC) doesn't guarantee a system is free of vulnerabilities. However, coverage is still widely regarded as a useful measure of software testing whilst algorithms like MC/DC are mandated within DO-178C to ensure adequate testing of the most critical (Level A) software.

The guidance for coverage with fuzz testing provided here is to adopt an approach that mirrors the intent of DO-178C coverage objectives whilst recognising that 100% MC/DC coverage is likely un-achievable with fuzz testing, within a reasonable time frame. Therefore consideration must be taken over Security Assurance Levels (SAL) of the security measures placed within the system under test in much the same way objectives are met for equivalent Development Assurance Levels. For example, when fuzz testing is targeting a security asset that has properties associated with safety critical DAL components, greater assurance will be required when compared to fuzz testing security measures with no associated safety hazard. Greater assurance can be achieved with increased coverage and an increase in the time the fuzz test is allowed to run.

It is also recognised that it may not be feasible to measure anything other than statement coverage with the majority (if not all) fuzz testing capabilities. The reason being is that, and at least with the popular "grey box, path aware" fuzzing solutions, test-cases, that do not identify system anomalies and also do not find new paths of execution though the control flow, are not retained for coverage analysis. Ordinarily, and especially with fuzzing solutions built on top of tools like American Fuzzy Lop [9], the identification of new paths through the control flow is determined by an analysis of which instrumentation points (placed around basic blocks) were hit. This is effectively a measure of statement coverage at the Assembly language level. In order to measure anything like MC/DC we would need to run the analysis on every mutated test case (regardless of which paths they took; which is not straight forward as ordinarily test cases that don't find new paths are quietly thrown away after test execution).

Therefore, in the context of fuzz testing it only makes sense to consider statement coverage, and this should only be used as a measure of the quality of the starting corpus. A high level recommendation is therefore to adopt the following two phased strategy:

1. Work towards achieving a test-case starting corpus that achieves the desired level of statement coverage or that sufficiently targets the areas of the application where refutation testing is required
2. Execute a fuzzing campaign utilising the starting corpus and derive a formula for defining the stopping criteria of the campaign

In effect this requires the definition of a "Starting Criteria" and a "Stopping Criteria" and both should be clearly defined within the Refutation Test Plan. It should also be noted that it is likely that multiple fuzzing campaigns will be required to target different aspects of the system under test. When this is the case each campaign will need to be considered individually and appropriate "Starting Criteria" and "Stopping Criteria" defined on a per campaign basis.

It is recommended that the following two methods are considered for building a suitable starting corpus:

1. If available adopt an existing set of test cases that are known to achieve the desired level of coverage, or that are known to target the particular security aspects of the system that the fuzzing campaign is attempting to refute.
2. Manually build a starting corpus and then use a fuzzer to mutate the corpus into a wider reaching set of test-cases that either achieves the desired level of coverage, or can be seen to target the particular security aspects of the system that the fuzzing campaign is attempting to refute.

Method 1 above is seen as the most likely solution where Integration testing or Unit level testing has already been applied (or will be applied) to the System. This will be particularly true of the higher safety assurance level systems that are using a testing verification method for arguing safety assurance. However, some effort will likely be required to convert unit or integration level test-cases into test-cases suitable for fuzz testing, although it is reasonable to assume that some, or all, aspects of the conversion could be automated.

Once the starting criteria has been derived, established and accepted by the governing regulatory body the actual fuzzing campaign can start. However, in order to be able to sufficiently argue that the risk of the system containing vulnerabilities is acceptable we also need to clearly define the campaign's stopping criteria.

This formula will need to take into account multiple considerations (listed below) but will result in the following three fuzzing campaign requirements:

1. Starting Criteria has been met

2. Test hardware capability is achievable (measured in test executions per second)
3. Length of time the campaign will need to run has been calculated

Note that items 2 and 3 above are tightly coupled and could have been grouped together under the requirement "number of required test executions has been calculated".

In order to determine the formula required to derive the measurements above the following aspects need to be considered:

- maximum test-case permutations (if calculable)
- input data complexity
- system under test complexity
- achievable test executions per second (needs to consider hardware capability, number of processors, number of cores and factors like number of fork/executes per test case)
- mutations performed per test-case seed
- mutation algorithm strategies adopted (deterministic, random, structure aware etc)
- corpus cycles required (number of times every test case in the corpus has been through a complete mutation cycle)
- Security Assurance Level of implemented security measures being targeted

An example stop criteria formula is shown in the guidelines section for consideration 6.2 below.

A.4 Planning Considerations

The following aspects of the adopted fuzzing strategy for refutation (as described further in the sections below) should be covered within the PSecAC and per fuzzing campaign required:

- Mutation algorithm/s definition
  - A description of each algorithm adopted

        - algorithms built into any third party fuzzer mutation engines (for example if using AFL state the version of the tool used and the mutation phases adopted - i.e˙ deterministic / havoc etc)
        - any customisations to the algorithms (i.e˙ any sanitisation of the mutations)
        - any project specific algorithms developed (i.e˙ protocol aware mutations)

- Starting Corpus generation strategy
    - A description of the mechanism used to generate the starting corpus
    - A description of how coverage analysis will be performed (if applicable)
- Anomaly detection mechanism enforced
    - A description of the run-time checks used to detect vulnerabilities
        - run-time checks enforced by the programming language
        - additional third party vulnerability detection (for example AddressSanitizer)
- Test-case generation capabilities beyond test-case mutation, for example:
    - Symbolic Execution
    - “Magic byte" finding tools like RedQueen ([7])
- Stopping Criteria formula and resulting campaign requirements metrics

A.5 Guidance

1.1 What evidence can be collected to argue that a level of separation exists between verification and the refutation activity?

Fuzz testing is inherently different from traditional verification activities. Verification testing focuses on providing evidence that functional requirements have been satisfied. Fuzz testing does not focus on the correct functional behaviour of the system and instead is used to assess the robustness of the system when subjected to unusual operating conditions. In this regard it makes for a good tool for identifying software bugs (which may or may not become classified as vulnerabilities), particularly where the vector that instigated the failure path would not normally be considered by traditional verification test cases.

In addition, test case automation with no prior knowledge or influence of functional requirements is clear evidence that the activity is separate to verification. One method to achieve this form of separation is to divert the number of test decisions away from the test engineer and towards the fuzzing engine. Non-discriminating automation, and particularly where the automation engine has no prior knowledge of the input data format or the architecture of the system under test, reduces the risk that human preconditions can influence the test.

Fuzz testing involves the dynamic and automatic generation of test cases. This takes some of the onus of test case creation away from the engineer. However, the most effective fuzzing engines make use of a starting corpus which may, or may not, be manually generated. Each test case, from the initial set, is required to contain valid data that will take the system execution through to completion. In order to maximise the usefulness of the starting corpus each test case should aim to drive the test executions through differing paths within the control flow graph. This allows the fuzzer to gain a significant advantage with the goal of maximum coverage. Depending on the scope of the fuzzing session, and the nature of the system under test, it may be possible to create the starting corpus without having to understand the underlying architecture of the system under test (for example where an API is being fuzzed). However, in order to ensure the starting corpus is wide-reaching in terms of control flow it is expected that the test engineer will need to also understand the code base.

Human created starting corpuses increase the risk of human preconceptions influencing the decision over the data values. This can lead to an unwanted bias unknowingly influencing the test through prior knowledge of the system requirements. Divergence along all of the control flow paths from the starting corpus are within the reach of the fuzzer through standard test mutation. However, there is a risk that a control flow path is assumed to be secure and not targeted by a test case within the starting corpus. If the code base is complex it may be difficult for the fuzzer to randomly mutate the data such that all paths are executed.

A better alternative, to manual starting corpus creation, could be automated starting corpus generation. This requires the automatic generation of multiple values for test data type components within the test case. All permutations, of each generated value for each test data type, will then need to be created such that the starting corpus can be generated. Some fuzzing engines, particularly grey-box fuzzers, have a capability to reduce a starting corpus down to the minimal set required to execute maximum coverage. If this capability is not available (i.e. black box fuzzing) then a human created starting corpus may be the best approach.

The refutation test plan should therefore contain a description of the strategy used for starting corpus generation. This should include an argument over how influences over system requirements knowledge will not add an unwanted bias within the starting corpus generation.

1.2 What argument can be made over how the process for performing the refutation activity ensures the testing is performed from the perspective of an attacker?

It is recognised that whilst grey-box or white-box fuzz testing may yield greater returns than black-box fuzz testing there is a recognised benefit in additionally performing system level black box fuzz testing. In addition, it is unlikely that the attacker will have access to a grey or white box fuzz testing capability, however, black-box fuzz testing will likely be more in reach and therefore should not be ignored. In addition, specialist mutation algorithms applicable to a particular system, can play a key role in ensuring a wide reaching test input corpus is achieved. However, an attacker will likely only have access to a standard set of generic algorithms and a good way of adopting an attackers mindset is to also try and deploy the same attack methodology.

Whilst it is recognised that system specific mutation algorithms can help penetrate deeper into the control flow graph they should not be a complete substitute for generic mutation algorithms. Generic mutation algorithms are defined in this context as having no understanding of the nature of the test cases. System specific mutation algorithms are structure aware and are typically used to sanitise mutations and ensure the system under test doesn't immediately reject the test case at the system boundary. It is recommended if structure aware mutations are needed that they are used in combination with generic mutation algorithms. It is unlikely that the attack will have access to structure aware mutations - or at least the same structure aware mutation algorithms as the refutation tester. Attackers will however always be able to utilise generic mutation algorithms.

1.3 How does the activity deal with human preconceptions and unwanted bias within test cases?

For the most part fuzz testing involves automated test case generation, however, there are two aspects to fuzz testing that can include unwanted human bias within generated test-cases:

- Mutation algorithms
- Starting corpus generation

It is recommended that Fuzz testing strategies incorporate multiple mutation algorithms including random bit flip patterns. Test-case data structure aware mutations serve a purpose, however, should not be included in the strategy at the expense of random mutations.

A starting corpus generation strategy should be shown to attempt to achieve as much coverage of the system under test as feasible. If a starting corpus is too focused on a particular aspect of the control flow graph it may be difficult for the fuzzer to mutate the test-case seeds in such a way as to traverse the graph onto the wider reaching paths. However, this can be mitigated by ensuring a statement level coverage analysis is incorporated into the fuzzing session and the starting corpus generation.

2.1 What are the main goals for the refutation activity? What assurances is it trying to achieve and how?

The main goal of Fuzz testing is to automatically generate test-case inputs that try to execute the aspects of the control flow graph being targeted whilst detecting if the generated test-case inputs cause the system under test to crash or hang. However, fuzz testing is not completeness testing and cannot provide complete assurance that the touched code base is free of vulnerabilities, it can however demonstrate that the system is robust when subjected to a large sample of test-case input values.

Fuzz testing, as a refutation activity, is about trying to provide some assurances that the system is free from software bugs. Fuzz testing attempts to provide this assurance by identifying any software bugs that, when triggered by certain test-cases, cause the system, or part of the system, to inadvertently crash or enter an unknown state of execution. Identification of this behaviour is typically achieved via detection of a segmentation fault or through the monitoring of other application layers or operating system layer detected faults.

In addition, some fuzz testing mechanisms may also attempt to detect when a system has failed to complete its execution within a given timeout. This provides assurance that certain test-case data inputs cannot cause the system to enter lengthy processing states or become permanently hung or blocked which is the common aim of Denial of Service (DoS) attacks.

2.2 What unwanted behaviour is the activity trying to exercise in the system?

Fuzz testing will attempt to orchestrate scenarios that demonstrate that the system is robust and will continue to operate, within its functional specifications, under unusual operating scenarios.

In particular fuzz testing will generate test-case data that may or may not be correctly constrained and may or may not be considered valid by the system under test. The fuzz testing mechanism will then observe the system under test during the processing of the test-case and check for the undesired behaviour, such as crashes and process timeouts.

2.3 Is the activity targeting a known set of vulnerabilities/exploits or is the activity attempting to capture any possible vulnerabilities through techniques like brute force test case injection?

Fuzz testing is about the detection of data inputs that can cause the system under test to inadvertently stop executing, enter an unknown state of execution or overrun an execution cycle.

All of these unwanted behaviour states are caused through software bugs. Ordinarily common software bugs are not specifically targeted through fuzz testing and the identification of the software bugs that caused the unwanted behaviour are not known. However, there may be specific techniques, adopted within mutation strategies, that aim to trigger common issues such as buffer overflows or the raising of range check exceptions.

For the most part fuzz testing is targeting all software bugs that cause the system to crash or hang.

3.1 What are the known limitations of the refutation activity?

Whilst software tools exist to try and automate as much as possible the creation of a Fuzz testing session there are several system characteristics that make the technology harder to adopt. Typically fuzzing is well suited to systems that follow a traditional model of:

1. Receive input
2. Process the input
3. Stop

Systems that don’t fit this model will need to be altered. For example an application with a main permanent loop that processes messages from buffer queues will need to be altered such that the execution completes after a message is received and processed. This, however, is no different to the type of scoping involved in standard, more traditional forms, of unit level and integration level testing.

In addition, some fuzz based technologies rely on instrumentation of the system under test in order to provide guidance prompts to the mutation algorithms, others require complex test drivers, for example POSIX fork servers to efficiently spawn new test-case processes. When this "grey box" style fuzzing approach is adopted the system under test is no longer a true representation of the target executable code and different compilers may be needed to ensure the system can run within the required operating parameters of the fuzzing session.

Another consideration with fuzz testing is knowing when to stop. Fuzz testing mutation algorithms may become exhausted, particularly when the test-case input structure is considered simple or when a coverage analysis confirms that all inputs have been generated. However, it is more often the case that the complexity of the test-case input date is such that the maximum possible test case mutation permutations is vast. In this scenario the test engineer needs to have a well defined strategy for stipulating when the test can stop. Fuzz testing solutions that can offer up an automated capability to define complex stop criteria rules can provide a partial answer to this problem. It is expected that achieving acceptance of a refutation test plan that involves fuzz testing will require the documenting of a stop criteria algorithm that can be enforced by the fuzz testing tool. An example of a stop criteria algorithm is defined in A.5

When a fuzz testing session is terminated the coverage achieved should be assessed and, where applicable, additional manually created test-cases added to the starting corpus to drive the execution to the untouched areas of the targeted code base. The fuzz testing session should then be restarted. This process should be repeated until the required coverage for the starting corpus is achieved. Once we are satisfied we have met our starting corpus criteria we can commence the fuzzing campaign, which will be terminated when the stopping criteria is met.

3.2 What are the known factors that can stop the activity from completing its goals?

The greater the system complexity and/or data input complexity (as defined within consideration item 4) the less likely the fuzzing session will achieve the desired level of code coverage. A mechanism will be needed to measure either the complexity or the test-case coverage. Systems where fuzzing would be ineffective in achieving the desired coverage will likely need to be split up and fuzz tested as a series of units.

3.3 Are there any known hardware or system design architecture considerations that could affect the capability of the refutation activity (i.e. choice of Instruction Set Architecture, programming language or compiler optimisation).

Some programming languages are generally more well suited to fuzz testing than others. In particular languages with run-times that can be configured to include constraint checking make for good fuzz testing targets. Languages that support concepts like design by contract are also well suited. Languages that allow systems to enter unknown states through vulnerabilities such as buffer overflows are less well suited. An example of this is with the comparison of a standard Ada runtime, that will always identify buffer overflows, with a standard C runtime, that may, or may not, identify buffer overflows.

Some fuzz testing architectures require code instrumentation, however, most also support test execution using processor emulators. Fuzz testing tends to involve a complex test harness wrapper that needs to be built, or executed, alongside the system under test. This makes fuzz testing embedded systems, as generally seen within aerospace avionics, a particular challenge. Here the solution is often to rehost the application within an emulated environment that is representative of the target hardware and that it is accessible to a fuzz testing solution. However, and as noted within section 3.1, the embedded system may require alterations to allow fuzz testing to take place.

In addition, and particularly where a fuzz test harness requires a library component inaccessible to the target compiler, an alternative could be to cross-compile the application onto a host environment that is known to support the required component. An example is with the dependency AFL has on the implementation of POSIX *fork()* and *exec()* functions.

4.1 What is the definition of "system complexity" for the particular refutation activity?

Complexity is the key factor in determining the likelihood of the fuzzing session finding all of the vulnerabilities. In the scope of fuzz testing the term complexity can be split into two categories:

1. Complexity of the system under test

Measured by considering the volume and hierarchical depth of the decisions within the code base; the number of decision points plus the number of permutations of conditions making up the decisions. In addition conditions that may not traditionally be considered complex, around equality testing of data with wide ranging bounds (an example is floating point equality) is also considered complex in this context and this includes any conditions that make black box testing harder to achieve full coverage.

Other factors that may affect the complexity of a fuzz test include the level of concurrency within the architecture of the system and the number of external interfaces.

2. Complexity of the test case

Measured by considering the structure of the test case data, including the defined bounds of the individual scalar leaf nodes of the structure. In addition, consideration should be given to the protocol of the system that the data is adhering to and if the components of the data are loose or tightly coupled.

Another factor affecting the complexity of the test case is with the test case generation approach implemented within the fuzzer's mutation engine; fuzzing sessions using structure aware mutation engines will waste less cycles generating invalid data thrown away by the interface. However, custom mutators are intrinsically human-biased and it is advisable to always ensure they are not exclusively used; a pure structure aware mutation engine may fail to identify a vulnerability which can only be exposed through a test case containing invalid (unconstrained) data.

4.2 How does an increase in complexity affect the scalability of the activity?

An increase in complexity of the system under test will reduce the probability of the fuzzer reaching the desired level of coverage. Fuzz testing hardware considerations will also play a factor as the number of test-case generations required to reach the coverage target could be very high (millions or greater). Multi-core servers may be required to process fuzzing sessions on high complexity systems.

Scalability decisions will need to be a factor of the calculated complexity of the system and the hardware running the fuzzing session. Test-case generations per second will need to be complementary to the estimated total permutations of possible test inputs (if calculable).

Most fuzzing tools provide dynamic feedback during execution and where grey-box style fuzzing is adopted, test engineers will need to be mindful of the length of time that has passed since the fuzzer last found a new path of execution within the control flow graph. Last path time can be coupled with test case generation rate (hertz) to determine if the current approach is scalable to the system under test.

If the session isn't scalable there are two options available:

1. Complexity will need to be reduced
2. Test-case generation per second will need to be increased

5.1 What evidence can the refutation assurance activity provide to measure that unwanted behavior has been precluded to an acceptable level of confidence?

Fuzz testing approaches generally throw away test cases that did not result in unexpected system behaviour. This is by design as writing all generated test-cases to file is often not feasible. In addition, and where test-cases are dynamically created by mutation algorithms, this approach would result in an unwanted overhead that would have an adverse effect on the rate at which the tests can be executed. Therefore it is not always feasible to produce a set of repeatable regression tests that can be run to show assurance in the system.

Instead fuzz tests tend to record test-cases that cause the system to enter an undesirable state and test-cases that find a new path of execution through the control flow graph. The latter can be used to calculate code coverage metrics and the former can be used to show a previously identified vulnerability has been fixed. Where the coverage identifies gaps in the testing additional test-case seeds can be created and added to the starting corpus and the fuzz testing run again. This will need to be repeated until a suitable level of coverage is achieved.

5.2 What format will the activity use to record newly identified vulnerabilities?

Each fuzz test will be required to record the starting corpus and associated test harness such that the test can be re-executed. In addition, grey box fuzzers that record all test-cases that found new paths of execution through the control flow graph, should also save these test-cases into the starting corpus associated with the fuzz test.

It is however common for fuzz testing mutation algorithms to rely on random number generators to transform (mutate) queued test-cases within the corpus into new test-cases for test execution. Therefore fuzz testing sessions are often not truly repeatable. In some cases pseudo random number generators can be configured to use fixed seeds which can make aspects of the testing more deterministic. However, the usage of fixed seeds to produce deterministic fuzz tests is not a recommended strategy. This is because using random seed values is more likely to yield more interesting test case input values, achieve greater coverage and detect more vulnerabilities.

For fuzz testing there are two key factors in assurance evidence generation:

1. The ability to analyse the coverage achieved when executing test case inputs within the corpus.

2. The definition of a "stop criteria rule" used to argue that the continuation of a fuzzing session has reached the point of unacceptable dimensioning returns.

Section 3.1 provides an overview of the stop criteria concept.

In addition, test-cases that found unwanted behaviour within the system under test should be saved and later used to verify that the associated software bugs have been fixed.

5.3 How can the results of the assurance activity be reproduced and what information needs to be configured to achieve this?

Test-case files that result in undesired behaviour within the system will be recorded. These test-cases can be used to reproduce the vulnerability. In addition the fuzz test corpus, fuzz test harness and any execution environment settings will need to be retained, however, there is no guarantee that this will result in the same set of dynamically generated test-cases (as described in section 5.2).

5.4 What coverage criteria will the activity use and what are the known limitations for each case?

Fuzz testing is not bound to a particular code coverage algorithm. In addition, and assuming the fuzzer records all generated test-cases or test-cases that found new paths of execution through the control flow graph, it should be feasible to execute the recorded test-cases through the system under test independently of the fuzzing tool. In this sense multiple coverage mechanism can be utilised to record the achieved coverage through the system (i.e. Statement Coverage or Modified Condition / Decision Coverage etc).

The limitations of the coverage algorithm will therefore depend on the particular algorithm chosen, however, consideration will be needed over if, and how, the algorithm deals with factors like state incrementation leading to buffer overflow.

In addition, and as previously noted, it is expected that for many fuzzers it may not make much sense to record anything other than statement coverage. It is also recommended that this is only used to derive a formula for the starting corpus i.e. ensuring that the corpus meets 100% statement coverage

6.1 At what scope will the activity be performed (i.e˙ System, Unit or other)?

Like all software testing disciplines fuzz tests can be scoped using standard software boundary tightening techniques, for example function stubbing. Reducing the scope of a fuzzing session may be essential to perform an effective vulnerability test and there are multiple reasons why system level fuzz testing may be infeasible, these include:

- Concurrent applications where multiple system interfaces are simultaneously driven from multiple external sources
- Embedded applications that often enter a 'main loop' to process interrupts; anything failing to successfully terminate will be deemed as a 'hung process' by a fuzzer
- Applications that utilise closed source third party libraries
- Applications that contain protocol constraint checks of the input data under test (for example this may be required when a fuzz test mutation engine cannot sanitise the data's checksum)

Where feasible fuzz testing should be performed at the system level, however, the greater the complexity of the system under test the quicker this approach will lead to diminishing returns. Understanding the level of complexity involved in a fuzzing campaign is therefore essential for determining how to achieve the maximum effectiveness and ultimately how best to set-up, scope, execute, monitor and stop the tests.

As a minimum fuzz testing should be performed on identified attack vectors to the software system under test (i.e. socket connections) and ideally using a fuzz test harness that closely resembles that target environment.

6.2 What evidence can be provided to show that the activity has sufficiently tested the security measure/s commensurate to the Security Assurance Level associated with the measure/s?

Fuzzing focuses on the detection of software bugs. This includes “corner-case" design or implementation errors that are triggered via unusual data inputs; inputs that standard verification testing techniques may have failed to consider relevant.

Fuzz testing should consider the attack surface of the system under test as the fuzz test entry points and the associated fuzz test harness should be representative of the mechanism the deployed system uses to receive external data.

It is expected that there will always be a path from the actual system attack surface to the fuzz test injection point. In this sense each test-case injected into the system under test can be considered a potential attack vector. Potential security breaches or Denial of Service attacks can occur if the application can be triggered to transition into an unwanted state, an unknown state, a dead state or a blocked state. Fuzz testing can detect vulnerabilities that can cause these attacks via exploits deployed across the attack surface via associated attack vectors. In this sense any security measures that have been implemented to protect security assets associated with known vulnerabilities will also be subjected to the testing.

The following evidence should be collected to show the activity has covered sufficient scope commensurate to the identified security asset attack vectors:

- All test-cases that found vulnerabilities
- Evidence that found vulnerabilities have been fixed
- Description of the scope of the testing and the identified attack surface (the fuzz test injection point)
- Description of the mutation strategies adopted (the potential attack vectors)
- The coverage achieved by the testing
- Specific scenarios or a range of scenarios that any targeted security measures have been subjected to

In addition the starting and stopping criteria of the fuzzing campaign, as described within section A.3 needs to factor in the associated Security Assurance Level of the attack mitigating security measure(s) the campaign is targeting.

An example of how this could be achieved could be to multiply each variable within the derived stop criteria formula by the highest SAL number associated with the security measure(s) being targeted. Using this approach an example formula could look like:

$$\sum = (\frac{(\alpha \times \beta) + (\gamma \times \beta)}{\frac{\rho}{\beta}}) \times \sigma \tag{1}$$

where:

- α = Cyclomatic Complexity of the control flow

- β = Highest SAL Number associated with the security measure(s) being targeted
- γ = Input data structure complexity
- ρ = Achievable test executions per second
- σ = Some arbitrary number used to scale the result (i.e. 86400 = seconds in a day)
- $\sum$ = Hours of fuzzing time required for this campaign

An example of how this formula could be applied is shown in table 1. Here we can see that the required hours for the fuzzing campaign grows as we increase the Security Assurance Level. Note that this example assumes a starting criteria of 100% statement coverage.

$\sum$	β	α	γ	ρ	σ
104 hours	1	4	2	5000	86400
415 hours	2	4	2	5000	86400
933 hours	3	4	2	5000	86400
1659 hours	4	4	2	5000	86400

Table 1: Example derivation of required fuzzing time when factoring complexity, SAL and performance

Note that this example presents one potential solution to consideration 6.2, this has in no way been validated or agreed with a regulatory body as the correct approach. In addition no guidance is provided with this example on how to measure control flow complexity and input data complexity. It will be the responsibility of the reader to derive their own formula for calculating stop criteria requirements specific to their project.

6.3 What software representation level is the activity aimed at (i.e˙ source code or executable code or some other intermediate representation)?

Fuzz testing requires the execution of the system under test and source code has to be compiled into a target executable. This may, or may not, require a differing compiler to the target compiler and in some cases additional instrumentation may be required to be added at either the source code level or a lower level representation.

Fuzz testing is ultimately performed on executable code. The executable code may be the same as the actual deployed executable code or it may be a fuzz test specific required representation.

6.4 What are the limitations of targeting the activity on a particular software representation level, and how does this affect the system behaviour?

If the fuzz test can only be executed on a host environment then consideration will be needed over how the behaviour of the hosting processor compares to the target processor. As a minimum the following potential differences should be consider:

- Order or sequence of bytes within computer memory
- Performance implications of any embedded instrumentation
- Are the return values from any stubs (required to reduce the scope of the fuzz test) representative of the real system behaviour

The same considerations exist for aspects of verification testing and the same project guidance should be followed.

6.5 Is the hardware test-harness representative of the final implementation, and if not how does this affect the system behaviour?

This will need to be assessed on a case by case basis, however, fuzzing engines built on top of the popular American Fuzzy Lop (AFL) architecture will likely be limited to execution on POSIX based operating systems. If a fuzzing session can only be executed on a host version of a system hardware architecture there may be vulnerabilities introduced by the executable code generation of the target compiler, or the vulnerabilities within the target hardware, that will not be detected via host based fuzz testing.

6.6 Does the activity need to alter the software under test (i.e˙ insert instrumentation code for metric-related data collection) and how does the alteration affect system behaviour?

This will also need to be assessed on a case by case basis, however, it is not uncommon for fuzz testing strategies to require path awareness feedback instrumentation into the system under test. In addition some fuzzers will embed test-case execution constructs into the code base of the system under test to speed up the test-case execution time.
In this scenario it may not be feasible to fuzz test the actual application on its target hardware and this is likely to be particularly true of embedded flight software. Where this is the case the software will need to be ported to a native

platform either via cross-compilation and instrumentation, or through a emulation of its target environment. Fuzz testing is also reliant on the software under test executing to completion in as fast a time as possible. When fuzz testing at the system level the test engineer may have to accept that the code may need to be manually altered to ensure the exercised code completes an execution. For example an embedded system that enters a main loop and never terminates is not suitable for fuzz testing.
If target based fuzz testing is possible it should be used, however, if target based fuzz testing is not an option it is still considered useful to adopt an emulated or host based fuzz testing approach.
A description of how the strategy adopted affects the measurable performance of the system should be documented within the plan. Consideration should be given over if a drop in performance could affect time critical behaviour such that race conditions are introduced that are not reflective of the target release.

6.7 Can the activity test the boot process and verify the integrity is protected?

No. This isn't something that fuzz testing can achieve.

6.8 What aspects of the system can not be exercised by the activity?

Fuzz testing cannot identify security attack vectors exposed during a boot sequence (i.e˙ BIOS attacks / Boot Loader attacks). Fuzz testing cannot identify vulnerabilities exposed through hardware attacks (i.e˙ glitching, fault-injection, side channel), this includes any attack vector composed of a power fluctuation. There are no immediately identifiable software run-time aspects of a system that could not be exercised by fuzz testing. However, the more complex the system under test the harder it will be to identify all of the potential attack vectors.

7.1 How can the activity be configured to challenge the vulnerability evaluation?

Fuzz testing is a security verification activity and the main goal is to find software vulnerabilities. Any software bugs identified during a fuzzing session that are not currently identified within the vulnerability evaluation are a direct challenge to the evaluation.

7.2 What identified threat scenarios can the activity not cover?

Fuzz testing has the potential to identify vulnerabilities within standalone software applications, threat scenarios that involve the interaction of multiple systems can not easily be exercised. Complex scenarios involving double or triple failure conditions are best tested through other software verification techniques. Threat scenarios involving hardware attacks through methods including power fluctuations are better suited to other Penetration Testing techniques.

8.1 Has the activity been used in the field before?

At the time of writing fuzz testing is not considered to be commonly used within the aerospace and defence industries as a refutation activity. It is however more widely used within other industries, such as automotive and internet based client/server applications. This is expected to change with the introduction of the Airworthiness Security Process as an Acceptable Means of Compliance with cybersecurity related objects by various regulatory bodies, i.e˙ the Federal Aviation Administration (FAA) and the European Union Aviation Safety Agency (EASA).

8.2 Is this a mature approach of vulnerability identification or experimental?

Arguably fuzz testing is not tried and tested across all aerospace and defence projects however, it is gaining popularity. The technique is more commonly used in other sectors (i.e. automotive, internet of things).

8.3 If the activity involves a partner external to the organisation what evidence (certification packs etc) can the subcontractor provide?

To be assessed on a case by case basis.

8.4 How much human interaction does the activity involve and how can you demonstrate that the operators have the required level of experience for the associated tools?

The amount of human interaction will be based on the fuzzing strategy adopted and the fuzzing testing tool being utilised. The level of automation available will vary based on the complexity of the system under test and the capability of the fuzzing tool technology. Some solutions may offer a near-fully autonomous solution whilst others will require the construction of complex test harnesses, custom mutation algorithms and starting corpus.

Fuzz test harnesses are widely regarded as complex developments (when compared to standard verification test harnesses) and it is recommended that the chosen fuzz testing tool supports multiple layers of automation code generation, test execution and results collation.

Staff suitability should be assessed on a case by case basis, however, the more automated the fuzz testing solution the less experience the staff will likely need to operate the tools.

9.1 What other activities complement this activity and which other activities can make up for the known limitations of this activity?

It is expected that with complex systems fuzz testing may not generate sufficient test-cases to reach the desired level of code coverage.
Where this is the case other coverage measurable refutation activities (i.e˙ static analysis or formal proof) should be used to complement the approach.
Fuzz testing is not a replacement for penetration testing; although the two activities may cross over they tend to focus on different potential attack vectors. In particular penetration testing can be used to attempt to exploit a vulnerability within a boot loading sequence or a known vulnerability within a particular Operating System. In addition, penetration testing and static analysis focuses on the identification of known vulnerabilities whilst fuzz testing attempts to uncover any vulnerability (including previously unknown vulnerabilities).
Fuzz testing will likely yield little return on aspects of the system that have been formally proven to be absent of run-time errors. Therefore consideration should be given over the perceived benefits of fuzz testing high integrity aspects of the system that have been formally verified. For example if the system is implemented in the programming language SPARK and formally proved to be absent of run-time errors through the SPARK examiner it is unlikely that fuzz testing can provide any further security assurance.
In addition, other refutation testing activities like penetration testing should be used to cover all identified hardware attack vectors.

Acknowledgments

This research is part of the "High-Integrity Complex Large Software and Electronic Systems” (HICLASS) project that is supported by the Aerospace Technology Institute (ATI) Programme, a joint Government and industry investment to maintain and grow the UK’s competitive position in civil aerospace design and manufacture, under grant agreement No. 113213. The programme, delivered through a partnership between the ATI, Department for Business, Energy & Industrial Strategy (BEIS), and Innovate UK, addresses technology, capability, and supply chain challenges.

References

[1] RTCA. “"Software Tool Qualification Considerations Corrigendum 1"”. In: DO-330 (Dec. 2011), pp. 1–5.

[2] Safety-Critical Working Group RTCA SC-167 of RTCA and WG-12 of EUROCAE". “"Software Considerations in Airborne Systems and Equipment Certification"”. In: RTCA DO-178C (Dec. 2011), pp. 1–1.

[3] EUROCAE. “"Airworthiness Security Process Specification"”. In: ED-202A (June 2014). for the European Union Aviation Safety Agency (EASA), pp. 1–1.

[4] RTCA. “"Airworthiness Security Methods and Considerations"”. In: DO-356A (Sept. 2014). for the U.S. Federal Aviation Administration FAA, pp. 1–1.

[5] RTCA. “"Airworthiness Security Process Specification"”. In: DO-326A (Dec. 2014). for the U.S. Federal Aviation Administration FAA, pp. 1–1.

[6] EUROCAE. “"Airworthiness Security Methods and Considerations"”. In: ED-203A (June 2018). for the European Union Aviation Safety Agency (EASA), pp. 1–1.

[7] Cornelius Aschermann et al. “REDQUEEN: Fuzzing with Input-to-State Correspondence”. In: Jan. 2019. DOI: 10.14722/ndss.2019.23371.

[8] EUROCAE. “"Software Tool Qualification Considerations Corrigendum 1"”. In: ED-215 (Feb. 2021). for the European Union Aviation Safety Agency (EASA), pp. 1–5.

[9] Lcamtuf. *American Fuzzy Lop (2.52b)*. (accessed 23 Sep. 2019). URL: http://lcamtuf.coredump.cx/afl.