Overview of the BESSPIN Voting System

The BESSPIN Voting System (BVS) is a demonstration system being funded by DARPA's SSITH (System Security Integrated Through Hardware and Firmware) program. SSITH's goal is to develop new technologies that will enable chip designers to safeguard hardware against all known classes of software-exploitable vulnerabilities, such as memory errors, information leakage, and code injection.

The BVS demonstration system is intended to showcase the SSITH program's results by showing how the innovations of "securitized hardware" could be applied to a type of critical computing system that is personally familiar both in use and in importance to millions of Americans: voting systems. BVS is a combination of voting-system demonstration software together with prototypes of securitized hardware being developed by SSITH researchers.

This BVS Overview provide: an overview of the BVS itself and its availability for adversarial testing at DEF CON 2019; the high-level "win conditions" related to BVS operation; the threat model and boundaries of testing; specifics on win conditions with examples; and rules of engagement for testers's use of the DEF CON test environment and interaction with the staff operating it for the benefit of testers.

1. BVS Functional Overview

The BVS 2019 edition is a subset of the full demonstration system being developed for DEF CON 2020. The 2019 demonstration system focuses on a subset of a full voting system, with two specific computing devices: a ballot marking device (BMD), and a smart ballot box (SBB).

The demonstration BMD offers a familiar voting machine user interface to interact with a screen ballot, indicate choices, review ballot choices, finalize, and create a paper record of the ballot choices. Like BMDs in current commercial voting system products, the paper record is prepared for the voter to verify the correctness of the record, and then be cast and counted by an optical scanning ballot counting device. In some jurisdictions, the ballot would be cast into a voting place's ballot box, subsequently transported to a central facility, and the ballots centrally counted. In other jurisdictions, the ballot would be simultaneously cast and counted by "precinct count" system that scans and counts each ballot in the voter's presence, and stores the counted ballots.

The SBB's functionality is a small subset of an optical scanning ballot counting device. In the BVS 2019 demonstration, a tester acts as a voter, uses the BMD to make ballot choices and print a summary ballot, and inserts the paper into the SBB. The SBB's primary function is to scan the paper to find a 2-D barcode, decode it, and determine whether the decoded contents authenticate the paper as a legitimate ballot summary sheet from a BMD. If so, the SBB accepts the paper by scrolling it into the storage box; otherwise the rejected paper is ejected. The SBB does not perform a full scale analysis of the document, nor record the voter's choices, nor

tabulate the votes of the ballots it scans. The SBB's main function is to ensure that ballot boxes contain only legitimate countable summary ballot documents that can be tabulated later. Details of the SBB's operation are provided in Section 3.

The goal of the adversarial tester is to (1) tamper with the SBB's software via any software exploit that is in-bounds (see Section 7 for details), and then to (2) use the exploit to cause the SBB to fail in its main function, either by

- rejecting a legitimate ballot created by a BMD, or
- accepting an illegitimate ballot.

Section 7 has detailed information on "win conditions" including: testing boundaries for test activity, in-bounds exploits, and relationship to win conditions. However, at a high level, tester must create either of the above 2 events, **as well as** provide supporting evidence that the event was created by an exploit that is "in-bounds."

2. Testing at DEF CON 2019

The BVS 2019 edition will be available for adversarial testing at the 2019 DEF CON Voting Village. To enable testing, the BVS 2019 SBB will be available in three forms:

- At DEF CON Voting Village, there will be testable "target systems" with SBB software and securitized hardware that is intended to prevent specific types of attack that, if not prevented, would lead to testers achieving a "win condition" in the adversarial testing.

- The Voting Village will also include a "reference system" with the same SBB software running on similar hardware that lacks new security features, and is not expected to prevent winning attacks.

- The SBB software and hardware will also be distributed so that adversarial testers can inspect and recreate the entire process by which an SBB is created. The distribution will center on a downloadable virtual machine with the full build environment for SBB, including:
    - a Debian Linux image, RISC-V baseline documents and pointers to other documentation, RISC-V compilers needed to build the SBB, an anonymous clone of the repository containing source code and build tools, and standard tools used for network analysis and for binary analysis.

As a result of the latter distribution, testers can have a complete open-box environment with full information on SBB internals, used as needed to develop exploits against the SBB, with as much examination of the system internals and build process as desired. Detailed documentation will include descriptions of sample exploits that work on the reference system, but are expected to fail on the target systems. Testers can develop exploits in advance, then come to DEF CON, validate the success of their exploits on the reference system, and try their exploits on the target system.

3. Overview of BVS 2019 and Win Conditions

The BVS 2019 edition includes just the BMD and SBB. The target of attacks is the SBB only (with further limits specified in Section 7). The BMD is out-of-bounds. It is included as a system that can be used by testers for an interactive voting experience that produces paper ballots for interaction with the SBB.

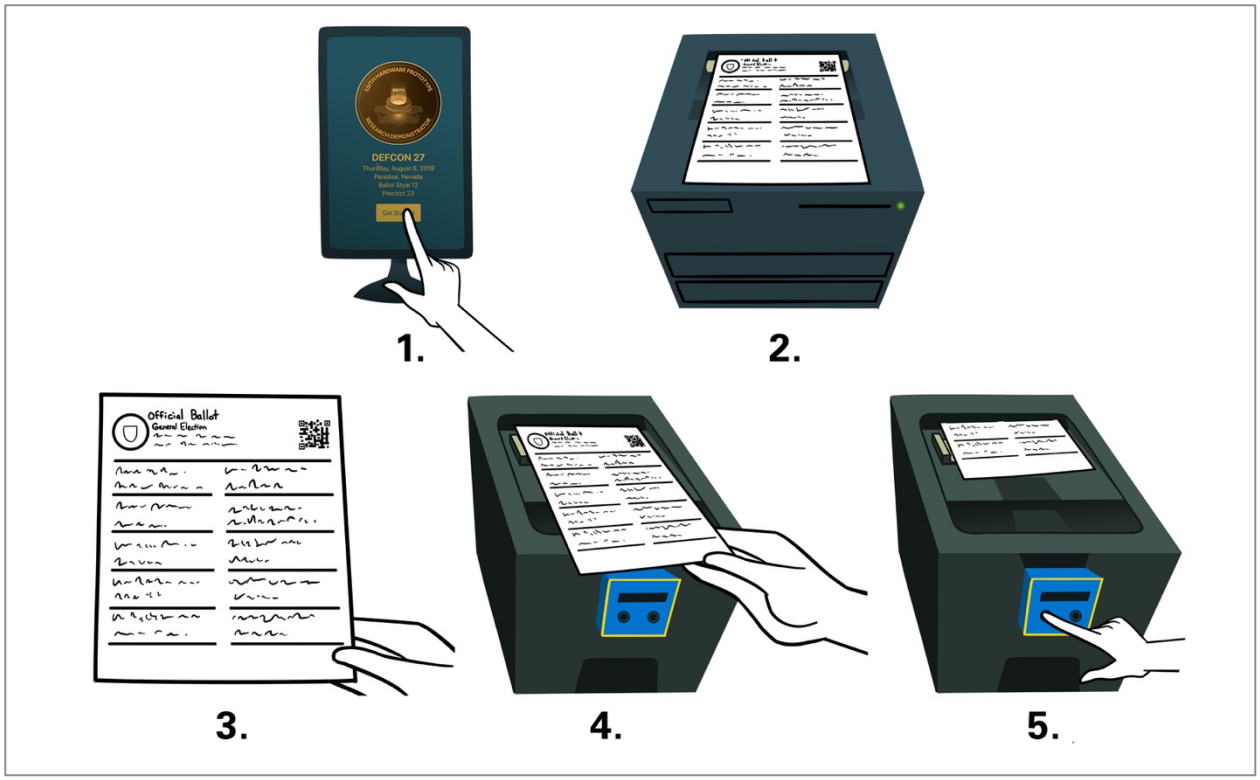

Figure 1: *Workflow for Ballot Marking and Smart Ballot Box Accept of Ballot*

Figure 1 shows one kind of normal workflow, where a person uses the BMD to produce a paper ballot, and then inserts the paper ballot into the SBB. The SBB validates the ballot, and asks the user whether to cast the ballot. The user chooses to cast the ballot, and the ballot is spooled through the SBB into its storage compartment. The SBB also creates log entries for each of these events.

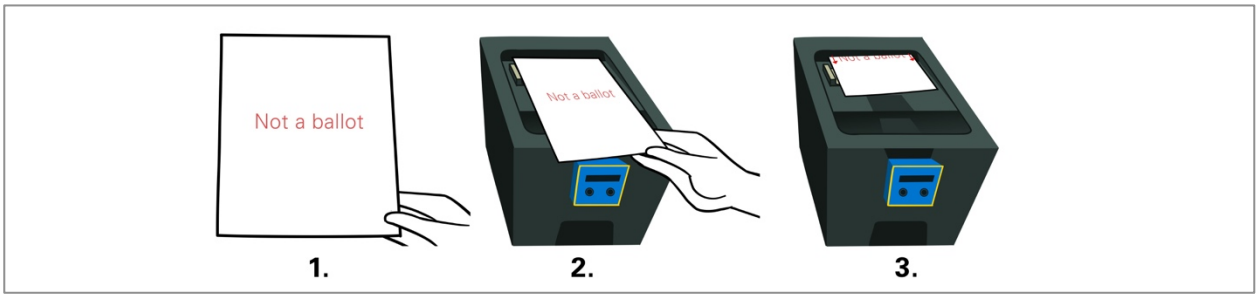

Figure 2: *Workflow for Smart Ballot Box Reject of Ballot*

Figure 2 shows another kind of normal workflow, where a piece of paper from any source other than a BMD is inserted into the SBB; the SBB's ballot validation check fails, and the ballot is ejected. The SBB also creates log entries for each of these events.

Not shown in the figures is the option for a voter to insert into the SBB a valid ballot, and then to choose not to cast the ballot, but rather to "spoil" it -- usually because the voter changed her mind about her ballot choices, wanting to spoil the current ballot and create a new one. A ballot so spoiled then becomes invalid, and if re-inserted into the same SBB, the invalid ballot will be ejected.

The primary visible indication of a possible win condition would occur when a tester produces improper external behavior of an SBB that violates the normal behavior in Figures and 1 and 2. The specific improper behavior would be either:

- the SBB accepting an invalid ballot, either created by something other than a BMD, or created by generating a BMD ballot and then using the SBB to spoil the ballot;

- the SBB rejecting a valid ballot just produced by a BMD.

In addition to the BMD and SBB, the DEF CON test environment also includes a separate Log Server, and a local area network to link the Log Server to the SBBs. The test environment function of the Log Server is to consolidate the log entries from the SBBs into one place, where test environment monitoring staff can view the logs. SBBs both record their logs locally, and send them to the Log Server via network communication. Log analysis by test monitor staff is required to validate a possible win condition, when a tester produces anomalous external behavior of an SBB (accepting an invalid ballot or rejecting a valid ballot). The Log Server is not part of the BVS 2019 voting system *per se*; rather, it is part of the test environment, included for examination of SBB log data in a manner more convenient than physically extracting them from an SBB unit via physical access.

Validation of a possible win condition depends on test monitors observing anomalous SBB behavior, and using log data to tie the behavior to a software exploit that is in scope. Non-software exploits are not a win condition, for example: using a BMD to produce a legitimate ballot; using a marker to deface the ballot so that it is not readable by the SBB; causing the SBB

to reject the BMD-produced ballot. Not all software exploits are win conditions, for example: attacking the BMD to cause it to produce what should be a valid ballot, but is in fact invalid and rejected by the SBB.

Specific boundaries of attack are described in Section 7.

4. Overview of Ballots

The BMD produces ballot summary documents, each of which lists the voter's choices for each of the contests and questions on the voter's ballot, but which omits the choices that the voter did not select. In addition to the choices, each BMD ballot also includes a 2-D barcode, the contents of which serve as an authenticator to the SBB. The SBB scans and decodes the barcode to obtain the data needed to determine if the ballot is authentic.

Figure 3: *Portion of a Sample Ballot*

Figure 3 shows a portion of an example ballot, with a header area below which is a list of the ballot's items -- contests and questions -- each with the voter's selection(s) if any. The header area contains identifying information about the election, ballot style; in the BVS 2020 edition the header might contain other information that a ballot scanner/counter device would need to tabulate the ballot.

The header area also contains a 2-D barcode (QR code). The BMD creates this barcode as an authenticator so that the SBB can recognize the ballot as having been created by a legitimate BMD. The authentication scheme is based on a Message Authentication Code (MAC) created with the AES standard encryption algorithm and a cryptographic key that is shared by the BMD and SBB. The SBB authenticates the ballot by re-creating the MAC with shared key, and comparing the result with the MAC encoded in the barcode. If the two MAC values are equal, the ballot is considered valid. Section 7 has more details on ballot authentication.

5. Overview of DEFCON Test Environment

Figure 4 shows the essential parts of the test environment:

- a BMD paired with the “reference system” SBB, running on ordinary hardware;
- a BMD paired with the “target system A” SBB, running on a current prototype of the securitized hardware;
- another BMD paired with the “target system B” SBB;
- the Log Server used by test monitors to examine log data produced by the SBBs;
- the LAN connecting the SBBs and the Log Server
- one or more attacker workstations also connected to the LAN.

Access to the LAN, for connection by attackers’ workstations, is provided for convenient access to an attacker toolset that is described in Section 7.

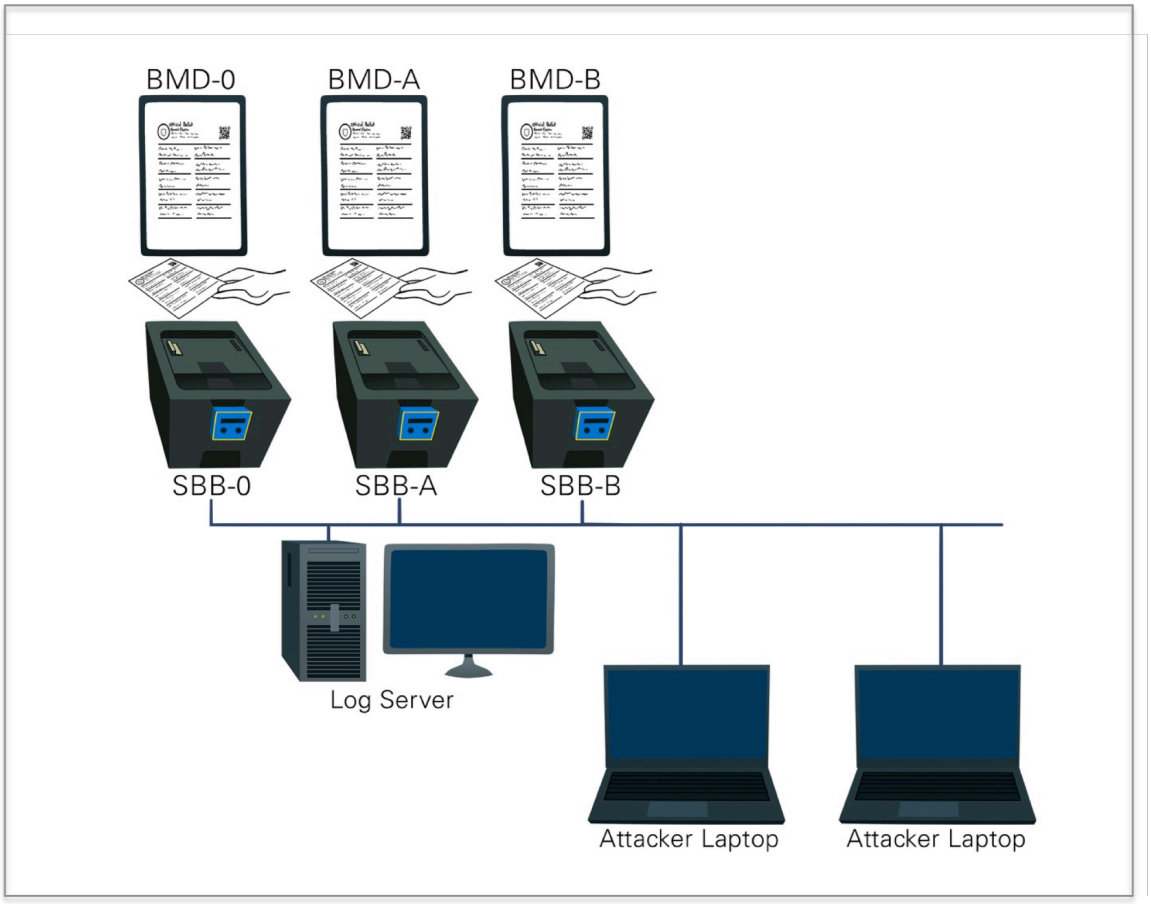

Figure 4: *The Elements of the Test Environment*

Figure 5 shows an additional view of the test environment, highlighting the part of the test environment that is the 2019 BVS, as distinct form non-BVS parts of the test environment. Also shown is the subset of the BVS that is the attack surface for the attacker, that is, the SBBs. Other parts of BVS, and the test environment, are out of bounds for attack.

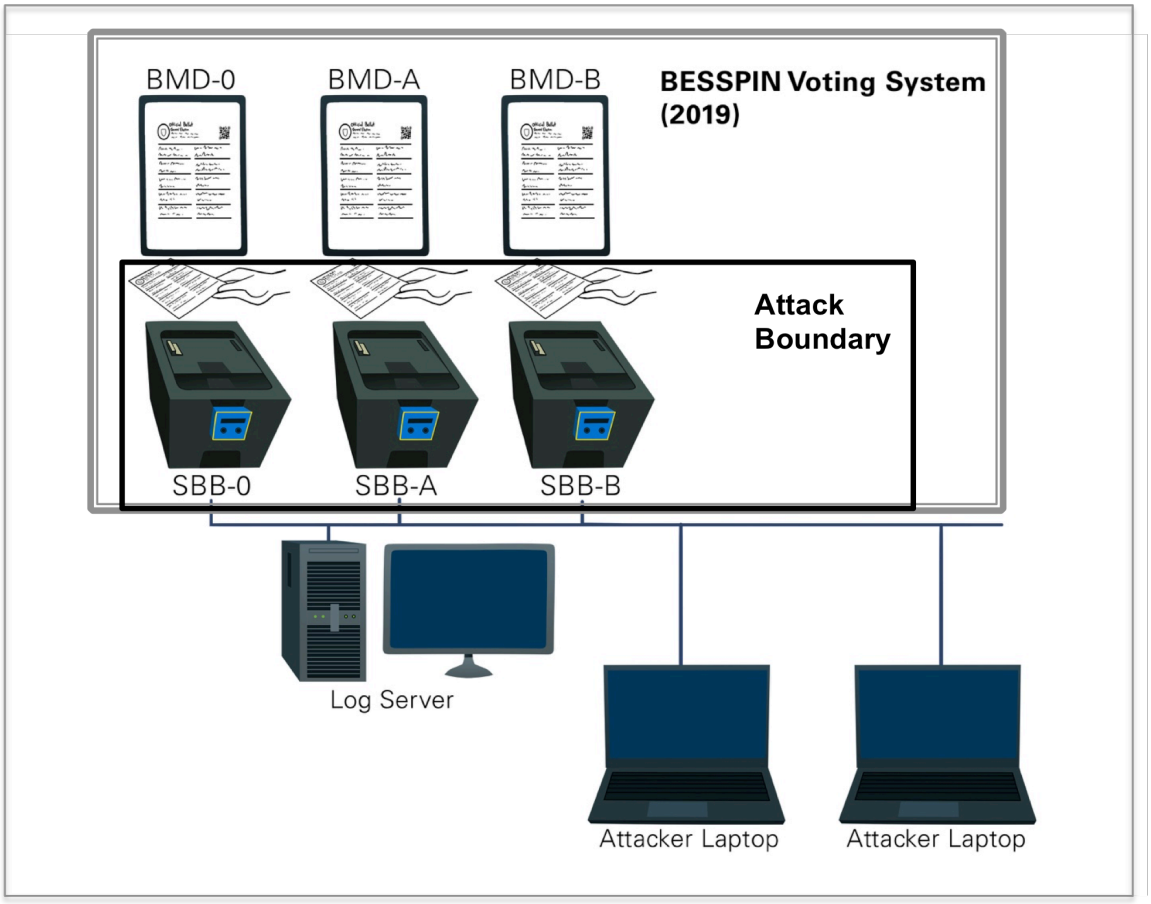

Figure 5: *Attack Boundary of the Test Environment*

Additional notes on the elements of the test environment shown in Figures 4 and 5:

- Each BMD is an ordinary personal computer and paper printer.
- The BMDs are “air gapped” with no networking, no communications in except for the physical peripherals, and no communications out except via the printed paper ballots.
- Each BMD is paired with a specific SBB. The SBB target system A will reject ballots produced by the BMD paired with target system B or produced by the BMD paired with the reference system. The SBB target system B will reject ballots produced by the BMD paired with target system A or produced by the BMD paired with the reference system.

- Attacks on the BMDs are out of scope; they may only be used to produce valid ballots that, if rejected by the corresponding SBB, might indicate a win condition.

- Each SBB is a combination of the following parts:
    - commodity parts for a scanner that is used to read the ballots, and for the printer/scanner feed slide;
    - a filing-cabinet-like enclosure to store the accepted ballots;
    - the primary circuit board, that includes the prototype securitized hardware, storage for the SBB's operating system and SBB embedded application software.

  Further details of the SBB's physical structure is available in published detailed technical specifications of BVS.

- In addition to the SBB embedded application software, each SBB's software also includes a web service that is not part of the SBB per se, but is included in the test environment to facilitate testers' activities. This tester's utility is intended to be used by a tester using an attacker's workstation (as shown in Figure 4) connected to the test environment's LAN. A tester can use the workstation's web browser to access a simple web interface that will enable the tester to attempt to read or write to any region of the memory of the SBB. This function provides testers with the option to skip a typical first phase of an attack that uses memory errors (e.g. buffer overflows) to tamper with the memory of a target system. Section 7 has more information on the tester's utility.

6. Threat Model

The goal of adversarial testing of BVS is to demonstrate that computing devices built with SSITH securitized hardware have fundamental protection mechanisms that prevent major categories of attack via software-exploitable weakness. For the BVS 2019 edition, the scope for attacks is limited to the one component, the SBB, that is built on 2019 prototypes of securitized hardware.

For BVS 2019 in specific, the threat model centers on an adversary who seeks the ability to tamper with an SBB unit, so that the tampered unit's altered operation could affect the outcome of an election. This threat model will be extended to other components of the BVS for the BVS 2020 edition, but the principle is the same: any voting system component, if successfully tampered with, could operate in a way that might affect the election outcome. In the case of the SBB, such tampered operation would be accepting illegitimate ballots or rejecting legitimate ones.

Adversarial testing of the SBB is intended to show that securitized hardware protects the SBB from attacks that change its behavior and put an election at risk. By extension to other types of systems than voting systems, securitized hardware could close off exactly the same avenues of attack that could be used by adversaries to endanger the mission of other kinds of systems.

As a result, the testing is limited to devices with SSITH hardware. Attacking other devices would not prove any useful concept for how SSITH technology provides new types of resistance to attacks via well known broad threat categories that result from software-exploitable vulnerabilities such as memory errors, information leakage, and code injection. Preventing these types of the attack is the goal of the SITH program.

For similar reasons, the BVS threat model excludes categories of attack that could disable a SBB unit, but would not cause it to continue to function after being compromised, e.g., functioning in a way that could stealthily accept illegitimate ballots for later counting.

Such excluded disabling attacks include physically disabling a unit, or creating denial-of-service conditions that prevent it from operating normally. Disabling an SBB unit is not a win condition.

Furthermore, a win condition includes not only the demonstration of an SBB with incorrect behavior, but also with the tester's ability to demonstrate that the success was a result of exploiting memory-error vulnerabilities despite the protections currently offered by the prototype hardware being developed on the SSITH program.

The next section provides specific boundaries for win conditions, while separately published technical documentation provides a list of specific Common Weakness Enumeration (CWE) items that are explicitly in scope.

7. Attack Boundaries for Win Conditions

The basic attack boundary is the limitation to attacks on the SBB, excluding all the other parts of the test environment shown in Figure 4: the BMDs, the log server, the LAN, attacker workstations.

In addition, physical disabling attacks on SBB units are out of scope, as are denial of service attacks. In addition to various forms of physically disabling an SBB unit, the following is a partial list of the more obvious DOS attacks:

- Tampering with the SBB's SD card that serves as the boot media. Erasing or corrupting the card's contents would prevent SBB startup. More careful tampering of the card, or substitution of an alternative card, would also prevent startup, because the hardware includes a trusted boot function that authenticates the boot media contents. An SBB unit will only start if the card's contents' hash value matches a previously authorized value.

- Tampering with SBB's internal storage unit to disable it or corrupt the contents.

- Tampering with the SBB's file system stored on the unit, which would corrupt the file system cause the SBB to crash or restart.

- Access to the SBB's I/O devices (indicator lights, motors, optical scanner, etc.) to cause them to be unavailable, inoperable, or unreliable in duties such as scrolling cast ballots into the storage container.

- Using access to the test environment LAN to conduct network DOS attacks. Of particular concern is network DOS that could prevent the log server from obtaining SBBs' log records that are necessary for test environment monitor staff to assess potential win conditions.

- Electrical power glitch attacks, which could disable the FPGAs and prevent further testing.

- Disassembly or reconfiguration of physical components.

Regarding the LAN, access to it is clearly in scope, for testers to use the attacker workstation described above. Snooping the Ethernet is also allowed. The contents and format of HTTP communication between SBBs and Log server are visible and may be observed, possibly with the aid of technical documentation. However, snooping data from the network, or injecting traffic into the network, are not by themselves win conditions, though it is possible that information gained might be a means to the end of creating memory errors or code injection.

8. Win Conditions and Sample Exploits

To achieve a valid win, a tester must meet two goals. First, the tester must be able to repeatedly demonstrate to the test facility staff their ability to interact with an SBB target system, and produce the required behavior of rejecting a valid ballot or accepting an invalid one. Second, the tester must demonstrate and explain the exploit that enabled the behavior, and show the exploit as being in scope.

To assist testers, some sample exploits will be provided, that show clearly in-scope methods that succeed in exploiting the reference system, but not the target systems. The following subsections provide some examples of such legitimate attacks.

Although many testers will choose to work in the familiar areas shown in the sample exploits (buffer overflows, return-oriented programming, etc.) testers are welcome to develop exploits in any of the 5 areas listed in detail in Appendix A as being specifically in-bounds.

8.1 Data Exfiltration of Cryptographic Key Material Enables Ballot Counterfeiting

Section 4 above described the method by which a BMD creates a barcoded authenticator that an SBB can evaluate as part of its decision to accept or reject a ballot. The method depends on the two devices privately sharing a symmetric cryptographic key. If a tester can obtain the key, then the tester can create a ballot with an authenticator that the SBB will accept. It is out-of-scope for a tester to obtain the key from a BMD, but very much in scope for the attacker to

obtain the key from an SBB by a method that does not violate other constraints described in Section 7.

One sample exploit to be provided is an exploit that accesses the key in the predictable readable memory location on the reference system. Exploit documentation will also include methods for creating an actual ballot from data that a tester believes is a valid exfiltrated key. To obtain the key via reading the memory, the testers' efforts will be reduced by using the attacker tool provided in the test environment. (See Figure 4.) Although this memory exfiltration exploit is expected to work on the reference system, but not on a test system, the sample exploit should be suggestive of other similar memory read-access exploits to test.

Note that there are several variations on the method of constructing a counterfeit of a BMD ballot. The ballot itself could be as well-formed as a BMD ballot; or could contain intentionally misleading data (such as a bogus or even ill formed timestamp) that would not affect the SBB's validation and acceptance. Any of these variations is a valid attack; the specifics of the content of the paper (besides the barcode) are not relevant, so long as the SBB accepts a paper artifact not created by a BMD, and so long as counterfeiting depends on the results of an in-scope attack on the SBB.

8.2 Software Modification Enables Modified Ballot Processing

In addition to providing convenient read access to SBB memory, the attacker tool also provides write access that can be used for attacks that include in-memory software modification. One sample exploit will be a write of a specific value to a specific region of memory that contains an instruction that is critical to the process of accepting or rejecting a ballot. For example, the SBB has a software state in which a ballot authenticator has been cryptographically verified, but the SBB has not yet completed the ballot intake process. A write to a specific memory location will modify the software so that the next time it executes, the SBB control flow will branch not to the software for accepting and storing the ballot, but rather to the software for rejecting and ejecting a ballot. As a result, the SBB software will (until memory is reloaded) reject ballots that are actually valid. (A similar attack could result in acceptance of invalid ballots.)

Again, while this sample exploit is expected to be re-creatable by testers to effectively compromise the reference system but not a target system, the sample exploit should be suggestive of other similar code over-writing exploits to test.

8.3 Return-Oriented Programming Enables Incorrect Ballot Processing

Another aspect of the attacker tool's memory write access function is the ability for the attacker to overwrite the stack return frame, and cause a return to the memory location of the attacker's choice. One sample exploit will demonstrate a simple example of such return-oriented programming: jump to a specific entry point in the existing software. For example, the SBB has a software state in which it waits for the user to indicate whether to cast or spoil a valid ballot.

The reference exploit can launched be at the point to cause a return to the existing software segment for rejecting the ballot as invalid.

As with other sample exploits using the attacker tool, the intent is to provide a “short cut” for testers to attempt to use the effects of a typical memory-abuse mechanism (such as buffer over-runs), but without the effort of actually finding a vulnerability and crafting an exploit to work through the vulnerability.

8.4 Arbitrary Execution of Injected Software

Another use of the attacker tool’s capabilities is to combine the methods of the 2 previous sections, to first write a specific open region of memory with executable code of the tester’s choice, and then return to execute that code. A sample exploit will demonstrate a basic form of code injection with effects visible via the attacker tool, but not actually creating a win condition. Testers will be able to use this reference code injection to craft more specific code injection attacks, the effectiveness of which can be validated on the reference systems, and tested on a target system.

For this and other aspects of the test environment’s support of testers, detailed technical specifications will be provided in an online public repository.

9. DEF CON Rules of Engagement

In the actual test environment at DEF CON, participants in the adversarial testing process are expected to follow specific rules of engagement with the staff who will be acting as monitors of the adversarial testing process. Testers are expected to have read this overview to understand the boundaries of testing activities, and to abide by them, including by refraining from testing denial-of-service attacks that could inconvenience other testers.

Testers are expected to first engage with test monitor staff, to be briefed on procedures to gain access, share access with other testers, and to demonstrate an understanding of the process for not only creating the visible sign of a possible win condition, but also for working with monitors to validate that the tester used an in-scope attack method.

Testers are encouraged but not required to prepare for testing by obtaining a wealth of information from an online public repository, including source code, build environment and tools, documentation of many kinds (including the tools provided for testers), and instructions for how to contribute to the testing portions of the repository.

For those testers who wish to contribute the project, the expected workflow is documented in detail in the public github repository for the SSITH Voting System Demonstrator.

Appendix A: In-Scope Common Weakness Enumerations

The Common Weakness Enumeration (CWE) is a category system for software weaknesses and vulnerabilities that is maintained in a repository (https://cwe.mitre.org) by a community of cyber-security professionals and supporting organizations. The most precise specification of in-bounds exploits is formulated in terms of specific CWEs. The following list is all of the CWE types that are in scope for the SSITH project. Exploits that match any of these CWE categories are in-scope, even though the current phase of SSITH efforts is temporarily limited to buffer/memory errors and information leakage/exfiltration.

Relevant to SSITH CWE class 1 (Buffer Errors)
CWE-118; CWE-119; CWE-120; CWE-121; CWE-122; CWE-123; CWE-124; CWE-125; CWE-126; CWE-127; CWE-129; CWE-786; CWE-787; CWE-788; CWE-823

Relevant to SSITH CWE class 5 (Information Leakage)
CWE-200; CWE-201; CWE-202; CWE-203; CWE-209; CWE-212; CWE-214; CWE-215; CWE-226; CWE-255; CWE-321; CWE-325; CWE-359; CWE-497; CWE-524; CWE-526; CWE-532; CWE-534; CWE-552; CWE-538; CWE-598; CWE-612; CWE-642; CWE-668; CWE-707

Relevant to SSITH CWE class 2 (Permission; Privileges; and Access Control)
CWE-799; CWE-307; CWE-837; CWE-284; CWE-287; CWE-288; CWE-289; CWE-290; CWE-291; CWE-293; CWE-294; CWE-257; CWE-259; CWE-260; CWE-261; CWE-262; CWE-263; CWE-301; CWE-302; CWE-303; CWE-304; CWE-305; CWE-306; CWE-307; CWE-308; CWE-309; CWE-345; CWE-346; CWE-384; CWE-521; CWE-522; CWE-523; CWE-549; CWE-592; CWE-593; CWE-603; CWE-620; CWE-640; CWE-645; CWE-798; CWE-804; CWE-836; CWE-285; CWE-613; CWE-732; CWE-862; CWE-863

Relevant to SSITH CWE class 3 (Resource Management)
CWE-401; CWE-415; CWE-416; CWE-562; CWE-590; CWE-672; CWE-706; CWE-761; CWE-762; CWE-763; CWE-771; CWE-772; CWE-825; CWE-911; CWE-188; CWE-395; CWE-476; CWE-468; CWE-588; CWE-690; CWE-706; CWE-822; CWE-399; CWE-400; CWE-404; CWE-405; CWE-610; CWE-664; CWE-669; CWE-694; CWE-774; CWE-913

(Relevant to SSITH CWE class 7 (Numeric Errors)
CWE-456; CWE-457; CWE-665; CWE-824; CWE-908; CWE-909; CWE-227; CWE-234; CWE-475; CWE-559; CWE-560; CWE-628; CWE-683; CWE-685; CWE-686; CWE-687; CWE-688; CWE-480; CWE-481; CWE-482; CWE-486; CWE-595; CWE-597; CWE-783; CWE-1024; CWE-128; CWE-136; CWE-189; CWE-190; CWE-191; CWE-192; CWE-194; CWE-195; CWE-196; CWE-197; CWE-369; CWE-681; CWE-682; CWE-704