CS 526 Information Security: Assignment 7

John Ross Wallrabenstein (jwallrab)

November 19, 2010

1 Problem 1

Outline

We begin by addressing methods for evaluating Xinu with respect to the User Data Protection requirements of the Common Criteria documents. Secondly, we address the constraints necessary to secure an EAL6 certification. We are given that the Xinu operating system has already satisfied the requirements for an EAL5 approval, we only address the *additional* constraints that must be satisfied to obtain a rating of EAL6. In the interest of brevity, we only list the major section heading, rather than the specific requirements we address (e.g. ADV_IMP.2 rather than ADV_IMP.2.X_i). It is trivial to take the set difference between the EAL5 and EAL6 requirements. We focus on discussing the specific tools, methods and approaches an evaluator would take to certify Xinu at the EAL6 level. For clarity, our crucial points are *emphasized*, uld or printed in **bold face**.

User Data Protection

Access Control Policy

The model most suited for evaluating Xinu with respect to the Access Control Policy requirement is the **Take-Grant Model**. That is, the Take-Grant model provides the necessary discretionary access control modeling language to evaluate the Xinu operating system. Clearly, the Take-Grant model requires the discretionary access control policy be applied to the actions over all subjects and objects. Thus, if an action is permitted by the Xinu OS and not by its corresponding Take-Grant model representation, the requirement has not been fulfilled.

Access Control Functions

The discretionary access control policy (in this case, Take-Grant) governs the set of permissible operations with respect to the subject and object's identity, based on individual or group membership. Group memberships can be modeled through Take-Grant as special objects. Subjects are given *take* rights over group objects when they are members, and the group object has rights over all entities in the corresponding group. In this way, the Take-Grant discretionary access control policy models the ideal functionality of the Xinu operating system. Should an action be permissible in the Xinu operating system that is not permissible under the corresponding ideal Take-Grant model, the requirement has not been satisfied.

The requirement that the discretionary access control policy enforce access based on identity and group membership follows naturally from our previous discussion of the application of the Take-Grant model in the ideal case. Specifically, if the Xinu operating system is modeled under the Take-Grant model, the discretionary access control policies specified by this requirement are enforced. Clearly, if no rule (i.e. right) exists in the Take-Grant model, the default response is to deny access. This requirement can be evaluated, under the assumption that the Xinu

operating system implements the Take-Grant model, by evaluating the *can* · *steal* predicate. If the predicate returns true for any entity pair, then the requirement has not been satisfied.

Finally, the Take-Grant model naturally supports the final access control function requirements. If the Xinu operating system provides a series of specific actions that allow administrators to override the discretionary access control policy, then the first requirement has been satisfied. We will not elaborate on this point, as the specific actions are to be assigned by the developer. Clearly, the Take-Grant model allows users to specify and control sharing of named objects based on identity and group membership. Using the previously mentioned method, a user may give the *grant* right to a group or singular identity over entities which they are authorized to do so.

Residual Information Protection

With respect to ensuring that information content of a resource is made unavailable upon deletion to all objects, the approach for evaluation is not clear. The NSA and Department of Defense state that the only acceptable method of information removal is *degaussing* the physical drive with a powerful magnet. Further, the preferred approach is physical disintegration or mutilation beyond recovery. Clearly, such actions are beyond the capabilities of Xinu. However, we discuss two approaches for addressing this issue. The first is the most basic, and is provided by most modern operating systems. When a file is deleted, the link to the data blocks is removed from memory, preventing (the overwhelming majority) of users or applications from reading the deleted information. However, it is possible that a user or process may search free memory in an attempt to recover the deleted information content. This leads to our second approach: secure information deletion. This procedure provides several levels of security based on the number of rounds and entropy provided when overwriting the deleted information. The Department of Defense requires a minimum of 40 rounds, using a randomized algorithm to obscure previously written bits on the disk. That is, a series of (pseudo) random elements, $s \in \{0,1\}^*$, are generated and written over the deleted information content r times, where r is the number of rounds specified in the requirements. However, this is clearly a gradient of security, with fewer rounds providing reduced security, but increased performance. The Xinu operating system should be evaluated based on the level of security required in the specifications, and the capabilities of an operating system given widely available hardware.

ADV_IMP.2

The evaluator must verify that the developer has provided a mapping between the TOE design description and the entire implementation representation, demonstrating their correspondence. This can be achieved using the software tool **Rational Rhapsody**, by IBM. This tool takes a top-down approach, and allows an evaluator to check the design. The tool guarantees consistency and completeness without interaction from the evaluator, and the subsystem-module mapping design is visually displayed. An alternate tool, **Enterprise Architect**, provides similar functionality.

ADV_INT.3

The evaluator must verify that the entire TSF is well-structured and is not overly complex. The unix operating system has a natural ownership based policy, which makes the **Take-Grant Model** a natural application. Given that we are dealing with a system similar to unix, any justification significantly more complex than the Take-Grant Model should be rejected. Similarly, any less structured TSF should also be rejected. While Take-Grant appears to be the most natural TSF for Xinu, any widely accepted, formal discretionary access policy will suffice.

ADV_SPM.1

To satisfy the formal security policy requirement, we require that the developer use a widely accepted, formal security policy. Given our argument for the **Take-Grant Model**, we suggest that this security policy be used to satisfy this requirement. Any extensions must be well documented, formalized, and shown to be both consistent and complete with respect to the chosen model. Other formal models, such as Bell-LaPadula, may be used. However, we feel that the most appropriate security policy is Take-Grant. The Bell-LaPadula model does provide some discretionary access controls, however it is primarily mandatory access control based. Other models may be rejected outright (i.e. Chinese Wall Model) if they are clearly not applicable.

ADV_TDS.5

To satisfy the design requirement of a semiformal description of each module, we recommend the use of **UML: Unified Modeling Language**. That is, UML provides a natural method for describing each module in terms of its purpose, interaction, interfaces, return values, and called interfaces to other modules. UML also provides fields for an informal, explanatory text string when necessary. Other methods, such as JavaDocs, do not provide a graphical representation of the interactions between different components by itself. The adoption of UML to satisfy this requirement reduces the work necessary for the evaluator, and presents the design in a graphical, sufficiently minimal descriptive representation.

ALC_CMC.5

To satisfy this requirement, we propose following **Lipner's Integrity Model**. In addition to combining Bell-LaPadula and Biba, Lipner's model was specifically designed for software development and deployment. To the best of the author's knowledge, a more fitting and widely accepted formal model does not exist. This model requires approval by a separate entity before commitment. That is, the *principle of separation of duty* must be followed. The UML model previously proposed will show how changes to a component will (potentially) affect other components in the system. To ensure that configuration items are maintained under the CM system, a bug-reporting system should be included. Further, a repository such as **CVS** or **SVN** would aid in both the reporting of bugs, as well as maintaining a version identity for each implementation representation. The evaluator should verify that the CM documentation for the specified version includes a CM plan, and that the CM plan describes the procedures used to accept modified or newly created configuration items as part of the TOE. If the above tools are demonstrated to be in use, then the evaluator can conclude that the CM system is being operated in accordance with the CM plans.

ALC_DVS.2

This requirement is satisfied by our recommendation for widely accepted, formal security policies; specifically those suggested in ADV_INT.3 and ALC_CMC.5. That is, the **Take-Grant Model** provides the security policy, and **Lipner's Integrity Model** satisfies the integrity requirement. As these models are only suggestions, if different models are selected the evaluator must verify that the models meet the security and integrity requirements.

ALC_TAT.3

The evaluator must verify that an implementation standards document is provided, and provides an unambiguous description and meaning for all conventions. Additionally, such documentation must be provided for all third party providers that collaborated on the project. The developer's (resp. third party's) implementation standards may be proprietary, but as long as

the implementation standards define a *consistent structure*, the document is acceptable. An evaluator is likely to cross reference the standards documents against the code. For example, the UML documentation will help to demonstrate that naming conventions are followed. Similarly, having sections of code that illustrate each implementation standard prepared will help with the evaluation process.

ATE_COV.3

To satisfy this requirement, extensive testing using a large number of inputs chosen from the full domain of possible inputs is necessary. That is, a **stress test** should be run to verify that the result of operations are consistent with the specification. Exhaustive testing is not possible; consider the *impossibility* of such a test on a function accepting inputs from the domain $\mathbb{R}$. However, **bounds checking** and **negative testing** should be used to verify a representative sample of inputs (valid and invalid) result in a system state consistent with the specification. The UML documentation can easily be extended to describe the domain of inputs a function or variable *expects* to accept, and examples of behavior on these and invalid inputs should be prepared.

ATE_FUN.2

The evaluator must verify that the developer has included an analysis of the test procedure ordering dependencies. That is, the order in which commands are executed is potentially relevant to their result. Technically, as long as the document is present, the requirement has been satisfied. However, particularly because an operating system is under evaluation, critical commands (e.g. *chmod*) should be extensively represented in the test procedure ordering dependency document to demonstrate compliance.

AVA_VAN.5

To satisfy the requirement that the TOE is resistant to attacks performed by an attacker possessing *high* attack potential, **third party penetration testing** can be performed. For example, **Coverity** offers penetration testing services (in addition to other system analysis). Additionally, a standard **Red Team / Blue Team** approach could be taken. Note that both **black box** and **clear box** approaches should be performed. That is, knowledge of the internal structure of the system may influence the actions of an adversary. To prepare for any eventuality (given the *high* attack potential), this bias can be removed through third party penetration testing both with and without knowledge of the internal structure. Finally, the developer should ensure that the computing and communication power of the third party is substantial. A high attack potential is assumed to have considerable resources at their disposal; any third-party that does not fully utilize the resources available to them has not successfully satisfied the penetration testing requirement.

2 Problem 2

We begin by presenting an overview of the basic security requirements a web server should attempt to enforce. Using these security requirements, we consider the logging and auditing necessary by each component to record and interpret security breaches in a meaningful manner.

Security Requirements

1. Connections from separate users should not be allowed to share information.

2. A failure in one connection should affect other connections to the least degree possible.

3. Authentication failures and successes should be logged.

4. After n authentication failures, access to the account requires additional authorization via phone or branch office.

5. Connections to the same account should not occur from multiple IP addresses simultaneously.
6. Connections from a new computer to an account should require additional authentication and a log message.
7. Connections from bank employees should only have access to accounts held by the employee's branch office.
8. All transactions that change the state of an account, initiated by employees or customers, are logged.
9. The front end should communicate only with the application engine, and the application engine should communicate only with the database.

2.a

Front End

The front end of the web server should spawn a new application engine thread, passing the request to the new thread. In this way, a failure in a connection is isolated from other simultaneous connections. As the front end of the web server is state-less, minimal logging information should be recorded. This increases the number of requests the server can accept, without sacrificing the ability to log potential violations. That is, as the socket is passed to the application engine, the connection thread has access to all information the front end is exposed to. Thus, the overwhelming majority of the logging is performed by the application engine. However, the front-end should log the time and date that the web server was started. Finally, in the event of a "soft" crash, the web server should log the time and date the system went down.

Application Engine

The application engine is responsible for maintaining state throughout the duration of the connection. Thus, the majority of logging occurs within this component. Specifically, the application engine logs authentication attempts, the $\langle user\_id, ip\_addr, time, success \rangle$ tuples, and the attempted, failed, and completed transactions that occur during the connection. If the connection originates from a new computer with respect to the account (e.g. different ip_addr or cookie), a log entry is generated. A log entry is generated whenever the threshold n failed authorizations for an account is met. Attempts by employees to access any account (in their branch office or not) are logged. Should a failure occur, a log should be generated recording the state of the connection and any information relevant to the failure. Finally, any action that changes the state of an account, initiated by an employee or customer, is logged to allow the transaction to be reversed. Optionally, if employees are allowed to assume a higher privilege level, these changes must be logged as well.

Database

As this is a financial web server, the ability to reverse erroneous or fraudulent transactions is paramount to the security of the system. Thus, we require that all database transactions be logged in such a manner that reversal is possible. That is, if an employee or customer is able to execute a transaction in error or with malicious intent, the system log should provide sufficient information to fully reverse the transaction. Although the reversal may not be possible (e.g. due to insufficient funds after withdrawal), the log allows the bank to recover from the failure in a meaningful manner.

2.b

The auditing system should support automated processing of the log files to detect suspicious activity, and common fraudulent activity. Ultimately, the goal is to detect violations of policy. While our stated policy is far from complete, it is sufficient with respect to the question domain. Thus, we reiterate the security requirements, and address how the auditing system analyzes the log

files to detect a violation of the stated security requirement. The auditing system adopted by the bank should attempt to follow the requirements of the A1 classification of the **TCSEC**. That is, the auditing system should have a minimal impact on system performance and be highly reliable. Minimal impact on system performance is palatable with respect to the client server architecture, and high reliability is desirable as sensitive financial information is being processed. We present audit mechanisms to support only those security requirements stated below.

1. *Connections from separate users should not be allowed to share information.* As the log files will contain $\langle user\_id, ip\_addr, time, success \rangle$ tuples, access to a separate account from the same connection (not from an employee) will be recorded. If the audit system detects that a connection from a customer accessed information from a separate connection or account (that is not a sub-account of the authorized account), a violation of security has occurred.

2. *A failure in one connection should affect other connections to the least degree possible.* The audit system should compare the failure logs from the application engine and the front end to determine if a failure in a connection (handled by the application engine) caused a failure of the entire web server. If such a failure occurred, a single connection affected other independent connections and a violation has occurred.

3. *Authentication failures and successes should be logged.* The audit system uses this information to verify that the account was not accessed (or attempted to be accessed) from suspicious IP domains. For example, connections to a U.S. citizen account from Nigeria would be flagged as a potential fraud violation.

4. *After n authentication failures, access to the account requires additional authorization via phone or branch office.* The audit system should analyze accounts with significant failed authentication attempts. This is designed to detect brute force password attacks against accounts. Audit rules could aggregate lists of IP addresses that attempt to access multiple accounts without success.

5. *Connections to the same account should not occur from multiple IP addresses simultaneously.* The audit system should detect suspicious activity; particularly a user attempting to access their account (sub-accounts excluded) from multiple IP addresses simultaneously. This is analogous to the fraud detection in use by major credit card providers.

6. *Connections from a new computer to an account should require additional authentication and a log message.* The audit system should detect failed attempts to access an account from an unknown computer, and flag them as potential fraud.

7. *Connections from bank employees should only have access to accounts held by the employee's branch office.* The audit system should detect attempts by bank employees to access accounts not under their branch office control. Such events could indicate an abuse of power, money laundering, privacy violations, or other actions in direct conflict with the security requirements.

8. *All transactions that change the state of an account, initiated by employees or customers, are logged.* The audit system should verify that transactions are *valid*. That is, a transfer of k units of wealth from one account to another should move exactly k units. The audit system should cross reference transaction logs from the application engine with those from the database to ensure that a vulnerability in one system does not go undetected by others.

9. *The front end should communicate only with the application engine, and the application*

engine should communicate only with the database. The audit system should detect the transfer of information between unauthorized pairs of the system. Database accesses originating from the front end, for example, should be detected and flagged as a potential presence of malicious code.

3 Problem 3

3.a

Hashing

The problem states that the malware *"places the malicious code in an audio file"*, so we assume that the audio files existed on the host computer before the presence of the malware. Given this, before the files were infected, they were valid audio files. The malware must alter the audio files to include the proper frequencies for the interpreter. Thus, if a hash of the audio file was made prior to infection, the change could be detected by the host computer. This is similar to the *TripWire* package, commonly used to monitor system critical files for changes. If the malware is assumed to have read and write access to the host drive, nothing prevents it from updating the hash of the original audio file with that of the infected file. Assume each audio file is hashed prior to infection with a keyed cryptographic hash function H_k, where the key to the hash is k. Store a hash of each audio file on the host drive as $H_k(a_i)$, where a_i is the audio file. The key k for the hash function must be stored on some external media, to prevent the malware from recovering it. Now, even if the malware could overwrite the original hashes, or generate new hashes, the key k is not present on the drive. Thus, when the user computes $H_k(a_i)$, the hashed values will not match allowing the alteration to be detected.

Logging and Audits

Assumption: The malware is only capable of altering the audio files, and installing the interpreter. In order for the malware to successfully "execute" an infected audio file $I(a_i)$, the file must be read by the interpreter. Thus, if all read accesses are logged, an audit of the log files will reveal that the audio file $I(a_i)$ was accessed. If the logs also contain invocations of media applications that would usually read audio files, a discrepancy will occur. That is, a read access to $I(a_i)$ will occur *without* a corresponding execution of a media application on the host. If the audit system knows to check for such discrepancies, then the presence of the malware can be detected.

3.b

Restricting Read Access

The malware requires read access to infected audio files $I(a_i)$ in order to execute instructions. Without read access, the interpreter cannot transform the audio frequencies into instructions. Thus, a host could prevent an infected audio file from being "executed" by limiting the read permissions for all audio files. A naive solution would be to restrict read access to media applications only, as they are the most common reason for accessing audio files. However, this prevents the user from copying or transferring the audio files, as the OS needs access to the files to do so. Rather, read access to all audio files could be restricted unless the user authorizes the action. That is, whenever a request to read an audio file is made, the user must approve that they wish to do so. To adhere to the principle of psychological acceptability, an authorized media application could issue a single request to read all audio files on the host drive upon loading. Thus, the user does not have to individually approve each audio file they wish to listen to.

Restricting Write Access

Assumption: All audio files were originally uninfected on the host system. The malware does not generate new audio files, but rather modifies existing files. Similar to restricting read access

to the files, the malware could be prevented by restricting write access to all audio files. If the malware is unable to alter the audio files to include frequencies corresponding to instructions, the interpreter is rendered useless. Require that all writes to audio files on the host system be approved by the user. When the malware attempts to write to an audio file, the user will (hopefully) realize that they did not initiate the write and deny the request. Again, to adhere to the principle of psychological acceptability, an authorized media application could issue a single request to write to all audio files on the host drive upon loading. This may be necessary to update the metadata stored in the audio files.