Security Policy Analysis using Deductive Spreadsheets[1]

Anu Singh, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott D. Stoller, David S. Warren
Dept. of Computer Science, Stony Brook University, NY 11794-4400
Email: {anusingh,cram,ram,stoller,warren}@cs.stonybrook.edu

Abstract

As security policies get larger and more complex, analysis tools that help users understand and validate security policies are becoming more important. This paper explores the use of *deductive spreadsheets* for security policy analysis. Deductive spreadsheets combine the power of deductive rules (for specifying policies and analyses) with the usability of spreadsheets. This approach is introduced with a simple example of analyzing information flow allowed by RBAC policies and then applied in two case studies: analysis of computer system configurations and analysis of Security-Enhanced Linux access control policies.

Keywords: Security policy analysis, Vulnerability analysis, SELinux policy.

1 Introduction

As information systems get larger, more complex, and more distributed, so do their security policies. The security configuration information in a single Windows XP system or Security-Enhanced Linux (SELinux) system [11] typically includes tens of thousands of items of information, much of it low-level. The problem is multiplied in distributed system with many interacting systems and services. Security policy languages are also getting more complex, in order to express complex organization-level security policies; this is exemplified by rule-based trust management languages, such as Cassandra [1]. In short, understanding the interactions and overall effect of security policies, and determining whether they ensure the desired high-level security goals, is difficult. As a result, analysis tools that help users understand and validate security policies are of increasing importance.

Desiderata for a Security Policy Analyzer. A useful tool for developing and exploring security policies should allow analyses to be specified easily, in a high-level language. Expressing analyses as deductive rules in a Datalog-like language [12] is a natural approach, adopted in several systems for policy analysis [15, 20, 14, 10] and program analysis [24]. Analysis results should be accessible through an easy-to-use interface. The system should be able to provide explanations of why a particular analysis result is a consequence of the policy. To enable the exploration of "what if" scenarios, the user should be able to make changes to the policy, and the analysis results should be updated and re-displayed interactively. In addition, the system should be able to highlight the analysis results that change as a result of the policy changes.

Electronic Spreadsheets for Security Policy Analysis. Our vision for a flexible and highly usable policy analysis tool is driven by the electronic spreadsheet, exemplified by Microsoft Excel. Electronic spreadsheets are enormously popular, for several reasons. Their interactive 2-D tabular interface makes it easy for users to organize, view, and manipulate the data (inputs and outputs of computations). The results of all computations are automatically updated after every change to the data or formulas. Moreover, computations

[1]This work was supported in part by NSF under grants CNS-0627447 and CCR-0205376, by ONR under grant N00014-04-1-0722 and by DARPA under SBIR grant W31P4Q-06-C-0146.

can be specified intuitively, by example. After specifying one instance of a computation (e.g., sum of the cells in a row), a user can specify, by copying and filling (pasting), that a range of other cells (the destination of the filling gesture) should be computed in a "similar" manner. This allows users to specify calculations without thinking about parameterized operations and loops, concentrating instead on concrete instances of a calculation. The easy-to-use interface and the direct interaction with data largely eliminate the distinction between developers and users.

This paper explores the use of *deductive spreadsheets* (DSS) [17] for development and analysis of security policies. Deductive spreadsheets combine the power of deductive rules (for specifying policies and analyses) with the usability of spreadsheets, and generally satisfy the above desiderata for policy analyzers. In addition, DSS-based security policy tools can be developed and customized more rapidly than specialized tools written in languages like C++ or Java.

A DSS, like a traditional spreadsheet, is a two-dimensional array of cells. However, columns and rows in a DSS are labeled by symbolic values. For instance, an access control matrix is naturally represented in DSS by labeling rows with subjects, labeling columns with resources, and storing a set of permissions in each cell. This highlights another feature of DSS: unlike a traditional spreadsheet (and indeed in other logical spreadsheets, e.g. [2, 9]), each DSS cell may contain a *set* of values. Values may be tuples, so the sets can represent relations. Accordingly, the formula language is extended with operations on sets and tuples. Thus, DSS formulas, like deductive rules, can define relations in terms of other relations. The semantics of a DSS is given by translation to Datalog [12]. We have implemented a DSS system called XcelLog, which is implemented as an add-in to Microsoft Excel™and uses the XSB tabled logic programming system as the underlying deductive engine. The results presented in this paper were obtained on the XcelLog system.

There have been other proposals to combine logic with the spreadsheet paradigm. They are discussed in Section 5. In short, our approach differs from the others in a fundamental way by supporting set-valued cells and meaningful recursive definitions.

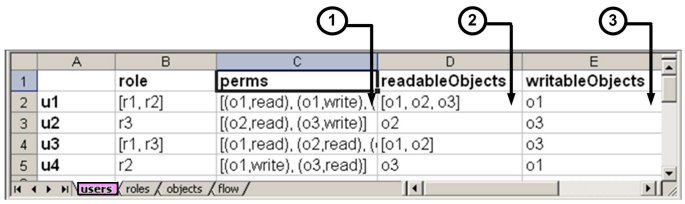
The remainder of the paper is organized as follows. Section 2 introduces the DSS paradigm. Sections 3 and 4 are case studies of using DSS to analyze computer system configurations and Security-Enhanced Linux access control policies, respectively. Finally, related work and discussion appear in Section 5.

2 An Illustrative Example: Analysis of RBAC Policies

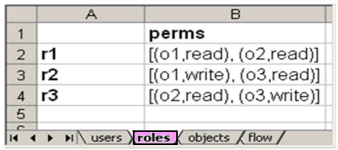
Role Based Access Control (RBAC) [19] is a well known model for expressing access control policies, and is the basis for access control in many commercial systems, especially DBMSs. Roles are useful intermediaries for relating users with permissions. Users are assigned roles *via* the *user-role relation*, and roles are associated with permissions *via* the *role-permission relation*. For example, a user may have permission to enroll in a course in an university if he/she is a student. The permission to enroll is associated with the *student* role, and a user gains the ability to enroll by being assigned the student role. The permissions of users can be computed by a join of the user-role and role-permission relations. In this introductory example, we do not consider other features of RBAC, such as role hierarchy and sessions.

We introduce the concepts and notations used in DSS by considering information flow analysis of RBAC policies. When a user can read from object o_1 and write to object o_2, we say that there is a (possible) direct flow of information from o_1 to o_2. Information flow analysis computes direct as well as transitive flows between objects.

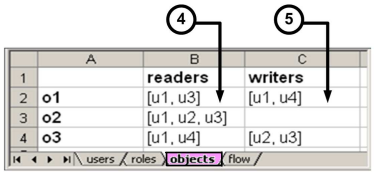
As in traditional (numeric) spreadsheets, a *workbook* consists of one or more *sheets*. Each sheet is a two-dimensional grid of *cells*. Each cell has a *value* which may be specified directly, or indirectly via a *formula*. A formula may refer to values in other cells in the workbook.



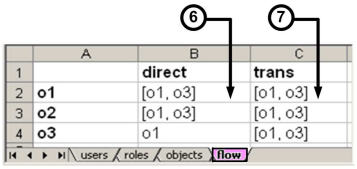
(a) Users



(b) Roles



(c) Objects



(d) Flow

1. `roles!(perms (role u1))`
2. `(perms u1){#2=read}[1]`
3. `(perms u1){#2=write}[1]`
4. `users!(~readableObjects o1)`
5. `users!(~writableObjects o1)`
6. `users!(writableObjects (objects!(readers o1)))`
7. `(direct o1) | (trans (direct o1))`

(e) Selected formulas used to construct the sheets in (a,b,c,d) above

Figure 1: Deductive Spreadsheets for the RBAC example

Part (a)–(d) in Figure 1 show screen shots of spreadsheets in XcelLog that express information flow analysis of RBAC policies. The formulas in selected cells of the spreadsheet are shown in part (e) of the figure. Comma separated lists of items enclosed in '`[`' and '`]`' denote sets. Singleton sets are shown without the enclosing square brackets. The `users` sheet (Fig. 1(a)) represents the properties of users for a given RBAC policy. The row names `u1`, `u2`, ... represent different users. Column `role` lists the roles assigned to each user. For instance, the cell at row `u1` and column `role` in the figure has values `r1`, `r2`, meaning that user `u1` has roles `r1` and `r2`.

Cell Values and Formulas. The elements of a set in a cell may be atomic values (e.g. `r1` or `r2`), tuples, or other structured values. Tuples are constructed with parentheses. For instance, consider the `roles` sheet in Fig. 1(b), which represents properties of roles. The value in cell at row `r1` and column `perms` is the set `[(o1,read), (o2,read)]`, which means that role `r1` has read permission on objects `o1` and `o2`.

Since DSS sheets have explicitly named rows and columns, we use a special notation for referring to cells. A formula of the form $s!(c\ \ r)$ is a reference to a cell at column c and row r in sheet s. The value of such a formula is the set of all values at the given cell. As usual, the sheet prefix $s!$ can be dropped if the reference occurs in the same sheet. Unlike traditional spreadsheets, however, the cell reference formula is naturally lifted to sets. For instance, if R is a set of row names, then $s!(c\ \ R)$ is a valid formula whose value is the union of the sets $s!(c\ \ r)$ for $r \in R$. The set of row names R may be specified by another formula. This construct enables simple specification of joins over binary relations. For instance, consider the cell at column `perms` and row `u1` in `users` sheet which lists the set of all permissions associated with user `u1`. The formula at that cell is `roles!(perms (role u1))`, and the value at the cell is the union of values of `roles!(perms r)` for all r in column `role` and row `u1` of the current sheet. The

formulas in other rows of column `perms` are obtained by copying and filling, which appropriately renames row and column references in the formula. Note the use of traditional spreadsheet metaphors to specify rules: first, the computation (a set-valued formula) is defined for a specific instance, and then the values for other instances are defined by replacing row/column names.

Tuple operations. Two operations, namely projection and selection, support the manipulation of tuple values in DSS. For instance, consider the formula to compute the set of all objects that are readable by user `u1`. Observe that `(perms u1)` is the set of all object/permission pairs to which `u1` has some access. Readable objects are selected using the formula `(perms u1)`{`#2=read`}, where the selection criterion is enclosed between '{' and '}'. The '`[1]`' at the end of `(perms u1)`{`#2=read`}`[1]` (formula 2 in Fig. 1) projects this set of pairs on the first component, thereby computing the set of objects readable by `u1`. The values in column `writableObjects`, which lists the set of objects to which an user has write access, is computed similarly.

Reverse Lookup. The formula `users!(readableObjects u1)` returns the set of all objects to which user `u1` has read access. Conversely, the set of all users who have read access to a specific object `o1`, which is represented by the cell `(readers o1)` in `objects` sheet (Fig. 1(c)) is computed using the reverse-lookup operation `users!(`$\sim$`readableObjects o1)` (formula 4 in the figure). In general, $(v\ \sim r)$ is the set of all columns c such that value v is in $(c\ r)$; similarly, $(\sim c\ \ v)$ is the set of all rows r such that v is in $(c\ r)$. Moreover, structured values may be used in a reverse-lookup operation. For instance, the formula `users!(`$\sim$`perms (o1,read))` also computes the set of all readers of object `o1`. The set of writers of an object, which is the set of users who have write privileges, is also computed similarly.

Recursive Definitions. When the formula in a cell x_1 contains a reference to another cell x_2 we say that cell x_1 directly depends on cell x_2. A cell x_1 is said to depend (directly or indirectly) on cell x_2 if (a) x_1 directly depends on x_2 or (b) there is an intermediate cell x_3 such that x_1 depends on x_3 and x_3 depends on x_2. DSS permits recursive definitions, so a cell may depend on itself. We illustrate this by encoding information flow analysis (see `flow` sheet in Figure 1(d)).

We say that information directly flows from object o_1 to object o_2 if there is some user u who can read from o_1 and write to o_2. Consider the computation of the set of objects to which there is a direct flow from o_1. The set of all readers of o_1 is given by `objects!(readers` o_1`)`. The set of objects that an user u may write to is given by `users!(writableObjects` u`)`. Thus the set of objects that some reader of `o1` may write to is given by `users!(writableObjects (objects!(readers o1)))` (formula 6 in Fig. 1).

Column `trans` in sheet `flow` represents the transitive closure of the `direct` flow relation, and hence expresses transitive information flow. It is computed as follows. In the base case, we say that there is a transitive flow from o_1 to o_2 if there is a direct flow from o_1 to o_2. Let O be the set of all objects to which there is a direct flow from o_1. The set O is the value of `(direct o1)`. Let O' be the set of all objects to which there are (direct or transitive) flows from some object in O. Thus O' is the value of `(trans` O`)`, that is, `(trans (direct o1))`. Note that O' contains the destinations of all flows that have more than one direct step; and O contains destinations of flows with exactly one direct step. Thus $O \cup O'$ is the set of all objects that are destinations of flows from o_1 using one or more direct flow steps. In DSS, the infix operator '|' is used to denote the union of two sets. Thus the formula in the cell at row `o1` and column `trans` (formula 7 in Figure 1(e)) is the union of the formulas corresponding to O and O'.

Relationship to Datalog. The meaning of recursive formulas is given as follows. Let x be a cell, and let f_x be the formula at x. Then the value at x is the *smallest* set that contains the value of f_x. This definition corresponds to the least model semantics of Datalog programs.

A set of spreadsheets defines a 4-ary relation: *sheet(Name,Row,Column,Contents)*, where *sheet*(S, R, C, E) is true if and only if E is in the cell at row R and column C in sheet S. For example, the cell at row `u1` and column `perms` in the DSS sheet named `users` in Figure 1(a) is defined by the Datalog rule:

```
sheet(users,u1,perms,X) :-
        sheet(users,u1,role,Y), sheet(roles,Y,perms,X).
```

XcelLog is implemented as a plug-in to Microsoft Excel™. It evaluates DSS formulas in the back-end by invoking the logic programming system XSB. DSS formulas are passed as parameters to a function called `DSS` that is implemented by the plug-in. When the value of a cell is needed during the evaluation of a DSS formula (e.g. when the formula contains a reference to the cell), the XSB side of the interface fetches the value from Excel. This architecture enables some cells to contain pure Excel formulas (e.g., statistical and numeric functions) while other cells contain DSS formulas.

Explaining Analysis Results. To help the user understand how the value (e.g., an analysis result) in a cell was obtained, XcelLog can highlight all of the cells whose value is used, directly or indirectly (transitively), in computing that cell's value; those cells are called its *precedents*. For example, consider the direct information flow from `o1`. To compute this cell, we need the values of the cell at row `o1` and column `readers` in the `objects` sheet, and also the values of every cell in the column `writers` of the same sheet. Transitively, we need all values in column `readableObjects` of `users` (and more). A cell's contents are completely explained by its precedents; in particular, if the cell contains a set, the cell's precedents completely determine which values are in the set, and which are not.

A more narrowly focused notion of precedent is sometimes useful as well. The *answer precedents* of a value v (in the set in a specific cell) are values (in specific cells) that explain the presence of v in that set. For example, to understand how there is an information flow from `o1` to `o3`, we can look at the answer precedents for the value `o3` in the cell in row `o1` and column `direct` in the `flow` sheet. The answer precedents for that value are the value `u3` in column `readers` of row `o1` in the `objects` sheet and the value `o3` in column `writableObjects` of row `u3` in the `users` sheet. XcelLog does not currently highlight answer precedents, but this feature is easy to add, since the underlying logic-programming system (XSB) can provide the necessary information. Similarly, XcelLog can highlight the *dependents* of a cell and could be extended to highlight the *answer dependents* of a value in a cell.

Incremental Re-computation. Spreadsheets provide an interactive environment in which data can easily be changed, and computed values are promptly and automatically updated. In the context of policy analysis, a user can modify the policy by editing the values in some cells, and immediately see updated analysis results.

When the value of a cell is changed, incremental recomputation proceeds as follows. First, the values of cells that are directly dependent on the changed cell are re-evaluated. Then any cell whose value changes due to re-evaluation triggers re-evaluation of its dependent cells. The re-evaluation process, which is based on the incremental evaluation algorithm of [18], continues until there is no further change in values. For instance, when user `u3` is removed from role `r1` and added to role `r2`, the cell in column `perms` in row `u3` of `users` sheet is first re-evaluated. Since the cell value changes to `[(o1,write)`, `(o2,read)`, `(o3,read)`, `(o3,write)]`, the other cells in the same row are re-evaluated. Note that the

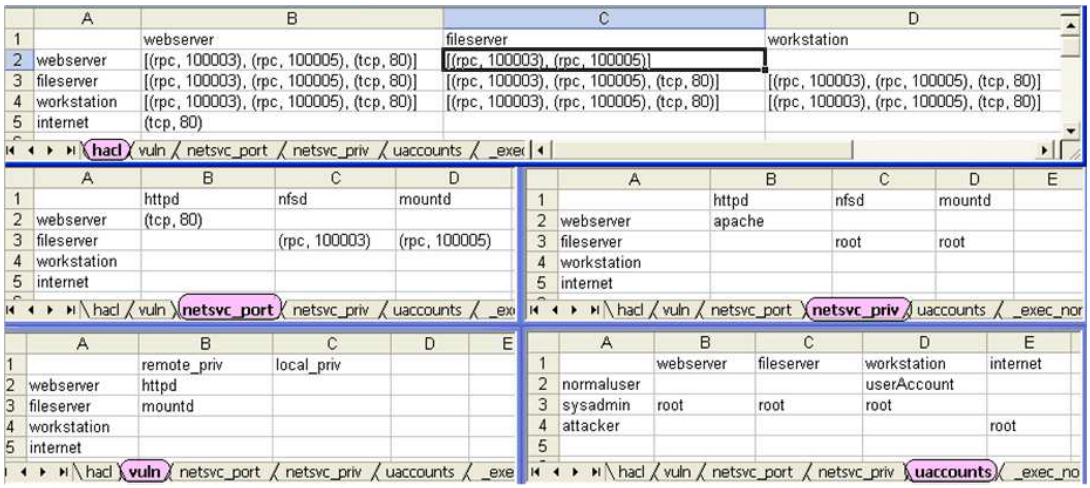


Figure 2: Vulnerability Analysis: machine and network configuration

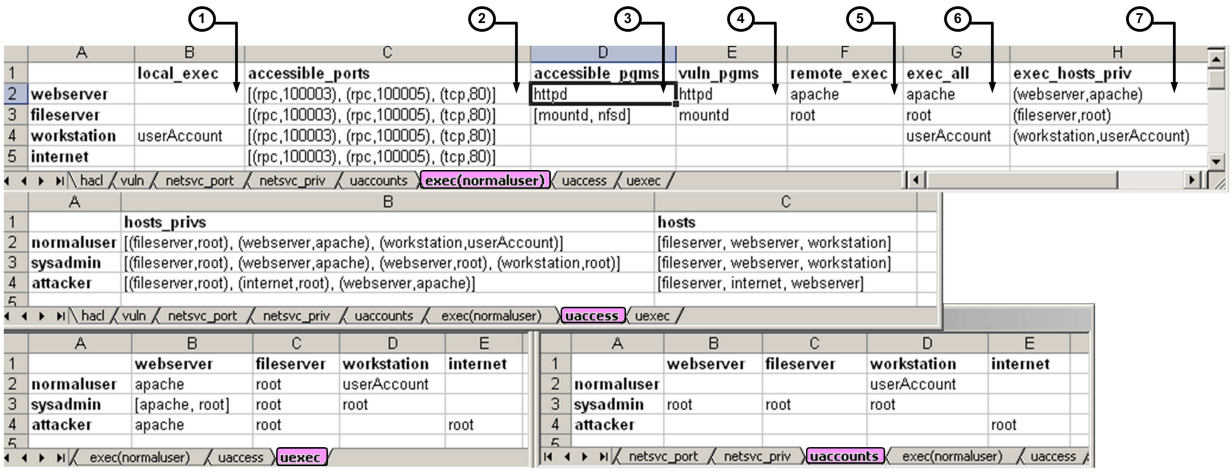
cell at column `writableObjects` becomes `[o1,o3]` while the cell at column `readableObjects` becomes `[o2,o3]`. This triggers the recomputation of all cells in `readers` and `writers` columns of the `objects` sheet. However, since the value at column `readers` of row `o2` in `objects` sheet does not change, the value at column `direct` of row `o2` in `flow` sheet is not recomputed. In large examples, incremental evaluation enables a user to efficiently explore "what if" scenarios by observing the effect of changing the values or formulas at specific cells.

Usability. We have described how DSS can be used to effectively encode a non-trivial analysis, using straightforward extensions to the popular spreadsheet metaphor. Policy analysis problems encoded this way inherit the usability benefits of the spreadsheet paradigm "for free". Firstly, the tabular layout can organize a considerable amount of information on the screen at once. Secondly, users can understand the results of calculations by navigating the precedents and dependents of cells. Thirdly, users can explore the effects of changes, since values in the spreadsheets are recomputed immediately whenever necessary when some cell values are changed.

3 Case Study: Vulnerability Analysis of Computer Systems

Vulnerabilities in an individual computer system may lead to exploitable vulnerabilities on an entire network of systems. MulVAL is a rule-based network vulnerability analyzer that assesses the impact of software vulnerabilities throughout a network [15]. This section illustrates how a DSS can be used to specify a part of MulVAL's analysis that deals with remote-access vulnerabilities.

Configuration Data. Figure 2 shows deductive spreadsheets that contain data about the configurations of different machines in a network. In the example, we consider a network consisting of a web server, a file server, and a workstation; machines outside of our network are abstracted into a single machine called "`internet`". The network configuration (firewall rules) is specified by `hacl` sheet (top sheet in the figure) which specifies the services (protocol/port pairs) on a machine that can be accessed from another machine in the network. For instance, in the sample configuration, the contents of cell at row `webserver` and column `fileserver` indicates that `webserver` can contact services on `fileserver` using `rpc`



(a) Deductive spreadsheets to compute acquired access privileges

1. `uaccounts!(webserver normaluser)`
2. `hacl!(webserver (uaccess!(hosts normaluser)))`
3. `netsvc_port!((accessible_ports webserver) ~webserver)`
4. `(accessible_pgms webserver) & (vuln!(remote_priv webserver))`
5. `netsvc_priv!((vuln_pgms webserver) webserver)`
6. `(local_exec webserver) | (remote_exec webserver)`
7. `(webserver, (exec_all webserver))`

(b) Formulas used to construct the sheets in (a) above

Figure 3: Vulnerability Analysis: computing acquired access privileges

on ports `100003` and `100005`. The services offered by machines in the network are represented by two sheets (middle sheets in the figure): `netsvc_port` which represents, for each machine and server program, the port and protocol used by that service; and `netsvc_priv` which represents the privilege with which the service operates on each machine. We consider three users: `sysadmin` who has `root` access to the machines in our network; `normaluser` who has regular access, denoted by `userAccount`, on the workstation; and `attacker` who has `root` access to some machine outside our network. These access rights are listed in the `uaccounts` sheet (bottom right in the figure). Finally, the sheet `vuln` (bottom left in the figure) lists, for each machine in the network, the set of vulnerable programs and whether the vulnerability can be exploited remotely or locally. The data for this sheet is obtained by scanning the machines for the presence of vulnerabilities listed in the CVE database.

The Analysis. Figure 3(a) shows deductive spreadsheets that determine whether any user can acquire more privileges than those explicitly allowed on a machine. The result of the analysis is the sheet `uexec` (bottom left in the figure) that lists the privileges that a user can ultimately acquire. The `uaccounts` sheet, listing explicitly allowed privileges, is repeated right next to it for contrast. The `uexec` sheet is constructed as follows. We begin with another sheet `uaccess` (middle sheet in the figure) to represent the privileges and access information in a form more convenient for subsequent computations. Column `hosts_privs` lists pairs of values (host,privilege) that represent the privilege a user (row label) may gain on a host. Column `hosts` simply lists the set of hosts to which a user has access. When we begin the construction, all cells in this sheet are empty. In order to fill the values, we construct a *template* sheet `exec(normaluser)` (top sheet in the figure) that collects the various attributes and privileges of `normaluser` on the different

machines.

Note that the template sheet is parameterized by `normaluser` and is used to derive attributes related to `normaluser`. The template sheet as well as the values in it can be instantiated to other parameters (e.g. `exec(sysadmin)`) when used in formulas. The formulas at cells in the first row of `exec(normaluser)` are shown in Figure 3(b). Column `local_exec` in this sheet lists the privileges that are explicitly granted to `normaluser`. Column `accessible_ports` lists the protocol/port pairs that are accessible to `normaluser`. For a given row h, this is computed by looking up all machines that `normaluser` has access to (from `uaccess` sheet) and finding the port/protocol pairs that are open from those hosts to h. The contents of the remaining columns are as follows. Column `accessible_pgms` lists the services that listen on the accessible ports; `vuln_pgms` column lists the accessible programs that run on the different machines of the network and have remotely exploitable vulnerabilities; `remote_exec` lists the privileges used by the vulnerable programs; and, finally, column `exec_all` lists the union of all privileges that `normaluser` can acquire.

The cell at row `normaluser` and column `hosts` in `uaccess` is defined to contain those hosts for which `normaluser` can gain some access, i.e., the rows for which `exec_all` column of `exec(normaluser)` sheet is nonempty. Note that the definition is recursive since `accessible_ports` column depends on the `hosts` cells of the `uaccess` sheet. We select row names whose contents are nonempty by forming tuples (column `exec_hosts_priv`), copying the column to (`hosts_privs normaluser`) cell of `uaccess` and selecting the first element of the pairs.

The attributes and access privileges of other principals is generated in the `uaccess` sheet by copying the first row and filling the remaining sheet with it. Finally, the first row in `uexec` sheet corresponds to the values in the `exec_all` column of `exec(normaluser)` sheet: it gives the privileges that `normaluser` can gain on the different machines. The privileges of other principals can be found simply by copying the first row of `uexec` and filling the remaining rows.

Discussion. This case study introduced the notion of template sheets which can be used to define the attributes of a specific instance. Values from template sheets can be copied into regular sheets and instantiated appropriately using the traditional copy and fill operations. The two dimensional grid of a spreadsheet is ideal for dealing with binary and ternary relations; template sheets provide a mechanism to deal naturally with higher-arity relations.

This analysis of multi-host remote-access vulnerabilities was expressed in DSS using the operations described in Section 2 plus template sheets. All classes of vulnerabilities detected by MulVAL [15] can be detected with a few additional sheets using these operations. The spreadsheet environment makes it easy to track the sources of vulnerabilities by navigating through precedents and dependents of cells. "What if" analyses can be performed by interactively changing the configuration data (e.g. the firewall rules in `hacl`) and observing the changes to the results, which are immediately recomputed.

4 Case Study: Security-Enhanced Linux

SELinux [11] extends the Linux kernel with a fine-grained mandatory access control (MAC) mechanism. The SELinux module is included in major Linux distributions, although it is turned off in the default configuration. The SELinux module enforces a security policy expressed in a language based on domain and type enforcement, extended with elements of RBAC and multi-level security. Much of the SELinux example policy [11] and reference policy [23] are devoted to fine-grained enforcement of the principle of least privilege for operating system processes and server processes, in order to strictly contain the damage that

can be caused by a compromised process. These policies are large (tens of thousands of lines), low-level, and difficult to write, understand, and validate. This motivated the development of several tools for development and analysis of SELinux policies [6, 5, 20, 4, 22, 14]. This section describes our prototype deductive spreadsheet for SELinux policies and compares it with other tools.

Overview of SELinux Policies. SELinux associates a *security context* with each resource (file, process, socket, *etc*.). A security context is a tuple $(user, role, type)$. A *type* is an abstraction analogous to a role: resources are organized into types, and subjects are granted permissions to perform operations on types (i.e., on the resources in the type).

An *access vector rule* indicates that resources (usually processes) in a specified type, called the *source type*, have specified permissions for the resources in a specified type, called the *target type*. A permission is identified by a *class* (i.e., a kind of resource, such as `file` or `process`) and an *operation* (such as `read` or `signal`). Types may have *aliases* and *attributes* (e.g., the `domain` attribute is given to types that can be assigned to processes). An attribute, when used in an access vector rule, represents all types having that attribute. The policy may also define a *never-allow* relation that describes accesses that should never be allowed; checking disjointness of the allow and never-allow relations can catch errors in the policy.

A policy also specifies the allowed roles for each user, and the allowed types for each role (more precisely, the allowed types for processes running in security contexts with that role). A security context $(user, role, type)$ is *consistent* if the role is allowed for the user, and the type is allowed for the role. The SELinux module blocks actions that would lead to the creation of an inconsistent security context.

SELinux Policies in DSS. SELinux policies in `policy.conf` format are loaded into our system by using a Perl script to transform the policy into comma-separated-value (.csv) format and then opening the .csv files in Excel (currently, some manual steps are also needed, but they could be automated). In our experiments, we analyzed parts of the SELinux example policy (version 1.1, for consistency with the experiments in [20]) and the SELinux reference policy (version 20061212) [23].

Sheets containing intermediate results and analysis results are computed from base data loaded in this way. The workbook provides many of the analyses in Gokyo [5], SLAT [3, 4], PAL [20], and NETRA [14], including the following.

Privilege Escalation Vulnerability. A *privilege escalation vulnerability*, also called a *write-execute (W-E) vulnerability*, exists when a non-administrative (i.e., less trusted) type has write permission for a resource, and an administrative type has execute permission for that resource [14]. As in the NETRA analysis [14], we introduce a `root` type with execute permission for selected types containing security-critical binaries, e.g., the type `su_exec_t`, which contains the `su` binary. We classify `root` as administrative, and all other types as non-administrative.

Our analysis found several W-E vulnerabilities in the SELinux reference policy, similar to vulnerabilities that NETRA found in the Fedora Core 5 targeted policy. For example, like NETRA, our analysis found a W-E vulnerability involving `apmd_t` (advanced power management daemon type). The vulnerability arises because `apmd` has the attribute `files_unconfined_type`, `su_exec_t` has attribute `file_type` (which corresponds to `attrib_001` in [14]), and an access rule gives `files_unconfined_type` write permission to `file_type`. Figure 4 shows some relevant parts of the workbook. The attributes of `apmd_t` and `su_exec_t` are visible in the `types` sheet on the left. The cell `(file_type files_unconfined_type)` in the `acv` sheet on the top right contains write access permission (not visible in the figure) besides other

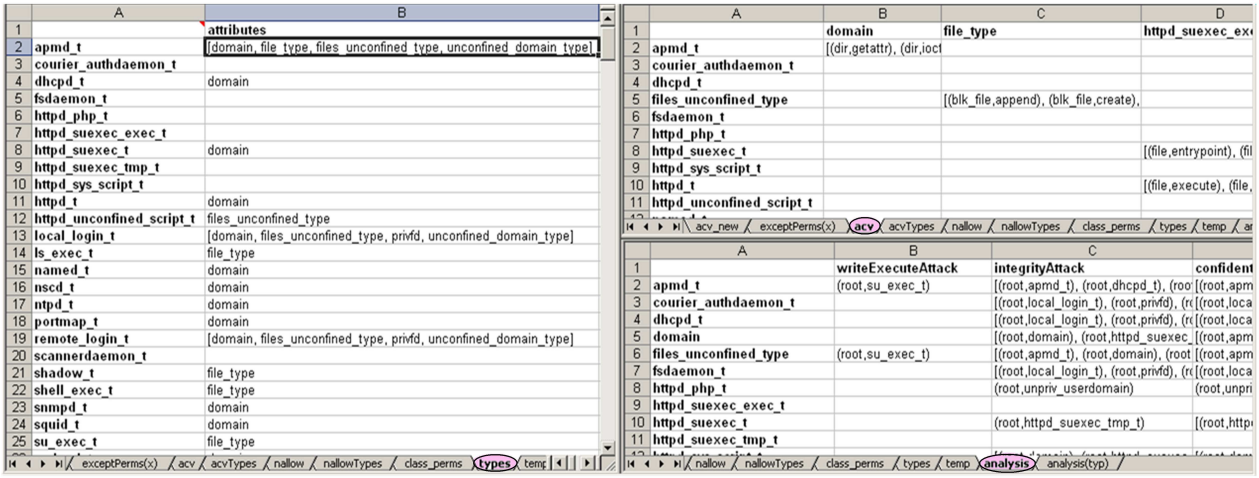


Figure 4: Write-Execute Vulnerability in SELinux Reference Policy

permissions on file objects representing the aforementioned access rule. The `(root, su_exec_t)` entry in cell `(writeExecuteAttack apmd_t)` in the `analysis` sheet on the bottom right indicates the vulnerability; specifically, it means that the non-administrative type `apmd_t` has write permission for `su_exec_t`, and the administrative type `root` has execute permission for `su_exec_t`.

Integrity Vulnerability. An *integrity vulnerability* exists when a non-administrative type s has write permission for a resource, and an administrative type t has read permission for that resource [14]. For this analysis, we gave the administrative type `root` read permission for selected types containing security-critical files, e.g., the type `shadow_t`, which contains the password file `/etc/shadow`. For example, we found an integrity vulnerability due to `apmd_t` having write permission to `shadow_t`, for similar reasons as above.

Integrity of Trusted Computing Base. This analysis checks for privilege escalation and integrity vulnerabilities, as above, with the administrative types taken to be the types in the trusted computing base (TCB) for SELinux proposed in [6]. As described in [6], the system designer may examine these potential vulnerabilities and eliminate the unacceptable ones.

Information-Flow Analysis. Information-flow analysis determines possible information flows between security contexts or types. Direct information flow is possible from a security context c_1 to a security context c_2 if a process with context c_1 has a write permission for resources with context c_2, or a process with context c_2 has a read permission for resources with context c_1 [3, 4]. The information-flow relation is the transitive closure of the direct flow relation. In the row for source context c, the cell in the `infoFlow` column contains the set of contexts to which information can flow from c. The sheet could be extended to provide specific paths along which information can flow.

We also developed similar but simpler sheets that, like NETRA [14] and Apol [22], calculate information flow between types, rather than security contexts, by ignoring constraints that involve the user and role components of the security context.

Information-flow analysis can be used in several ways. The user might want to simply examine the types from which or to which information flows for selected types of interest. The user might want to restrict attention to flows that do not pass through a specified type f that acts as an "information firewall";

this is easily supported by parameterizing the information-flow sheet by f and modifying the formulas appropriately. Information-flow analysis can be used to enhance other analyses. For example, a *transitive privilege escalation vulnerability* exists when there is information flow from a non-administrative source type s to another type s' and there is a privilege escalation vulnerability between s' and t. A *transitive integrity vulnerability* is defined similarly.

Policy Completeness. Jaeger *et al.* point out that *unspecified permissions*—i.e., permissions that are not in the allow or never-allow relations—reflect a kind of incompleteness in the policy specification [5]. *Policy completeness analysis* lists all the unspecified permissions, so policy developers can check whether the incompleteness is intentional and acceptable.

Usability. As mentioned earlier, precedents and descendants of cells, which can be highlighted in XcelLog, can be navigated to understand analysis results. However, finding highlighted cells in a large workbook could be tedious. XcelLog currently helps the user with this by marking sheets containing highlighted cells, but could be extended to automatically elide rows and columns that do not contain highlighted cells.

While spreadsheets are incrementally updated, it is sometimes useful to highlight all cells whose values changed as a result of the most recent edit operation. This is similar to showing answer dependents, but is more sensitive to the specific values involved: changing a value does not necessarily change the value of all its answer dependents.

After exploring changes to a policy in a spreadsheet, the user might want to make those changes permanent. It is easy to generate a `policy.conf` file that incorporates them, by saving the sheet as a .csv file and using a Perl script to convert the data to `policy.conf` format. A similar idea is mentioned in [10].

Performance. Our current prototype is usable but leaves room for performance improvement. For a workbook that implements all the analyses in PAL [20], including information-flow analysis (which involves a relatively expensive transitive closure) but excluding policy completeness, calculation of all analysis results for a subset of the SELinux example policy with all 271 types and 8700 access vector rules (43% of the total) takes about 5 minutes on a 1.7 GHz Pentium. Incremental re-calculation of all analysis results takes, for example, 1 second after deleting the attribute `domain` from the type `crond_t`; re-calculation takes about 1 minute after deleting all the attributes of `crond_t`.

Comparison With Other SELinux Policy Analysis Tools. Expressing policy analyses and requirements as DSS formulas is similar to expressing them as set formulas, like in Gokyo [5, 6], as regular-expression-like formulas, like in SLAT [3, 4], or as deductive rules, like in PAL [20], NETRA [14], and Lopol [10]. However, none of these other tools provide the usability benefits of DSS. In particular, none of them provides an interactive environment in which analysis results are incrementally updated as the policy is modified.

SLAT, PAL, and Lopol are command-line tools whose input and output are simply text streams. Apol [22] has a GUI for specifying the analyses to perform and displaying the analysis results, but it does not support editing the policy. NETRA does not support modifying the policy during analysis but provides a useful graphical output format: it generates derivations, in the form of directed acyclic graphs (DAGs), to justify and explain analysis results. A derivation provides the same basic information as a set of answer precedents, namely, indicating which policy rules contributed to the result, but presents it in a more structured manner.

None of these analysis tools, including our current spreadsheets but with the exception of PAL [20], work with policies in source-level format. A disadvantage of working with lower-level formats is that macros (with parameters), the primary abstraction mechanism in the SELinux policy language, have been

expanded; this can make the analysis results harder to understand. A useful direction for future work is to import source-level policies into spreadsheets, by translating macros into formulas in parameterized sheets.

5 Related Work and Discussion

There are several proposals for combining the spreadsheet metaphor with logic; some are surveyed in [13]. Knowledgesheet [2] and PrediCalc [9] are two recent ones that are related to DSS in their vision. Both of these approaches maintain the functional aspect of traditional spreadsheets, in that each cell still contains a single value. They extend traditional spreadsheets by allowing the user to specify constraints that partially or completely determine the value in a cell. Our approach differs from these in a fundamental way, in that we allow cells to contain sets of values and allow formulas with cell references to specify subset constraints. As as result, recursively defined cells do not make sense in their functional framework but are perfectly meaningful in our relational one. This is what really allows our spreadsheets to support full deduction.

Another interesting combination of rules and spreadsheets is ARulesXL (`http://www.arulesxl.com/`). ARulesXL allows users to define WHEN rules that specify cell contents using defined variables. The use of logic is interesting, but it retains the functional aspects of traditional spreadsheets and does not support recursive definitions.

There has also been recent work on extensions to the Excel spreadsheet that integrate *user-defined (non-recursive) functions* into the spreadsheet grid, rather than treating them as a "bolt-on" [8]. This work develops a way to specify user defined functions visually with a spreadsheet. But each cell still possesses a unique value.

Quantrix [16] has developed spreadsheet technology for compact representation of single-element cell data and manipulation of various dimensions to ease viewing of data under different representations. It will be useful to explore and generalize Quantrix's technology from single-element cell to sets for reducing the sparseness of data and facilitate multiple views of data in DSS.

Deductive spreadsheets might be viewed as a visual interface to a set-based language. While there have been several proposals for more expressive programming languages that support set specifications [7, 21], our focus is less on the power of the underlying language and more on its presentation and usability in the tabular spreadsheet form.

Deductive spreadsheets can be regarded as a different way of presenting the relational tables of a DBMS. A table view treats all components of a relation symmetrically, while the spreadsheet view introduces asymmetry, by choosing components to use as row names and column names, and tupling the remaining components. The DSS view is sometimes more compact, because a cell in a DSS may contain a set, while each entry in a DBMS table is an atomic value.

We conclude with some possible extensions to XcelLog, the current DSS prototype, that will further improve its usability. Currently, cell-level dependencies are considered when constructing explanations. By refining this cell-level granularity to the granularity of individual values in the set in a cell, more detailed explanations can be constructed. Another useful extension would be to allow the user to explicitly define sets of row or column names and then automatically extend a sheet (including generation of formulas for the new cells) when its row or column set is changed. For some policy analysis problems a Datalog encoding may already be known. In such cases, the initial development of DSS encoding can be simplified by generating it automatically from the Datalog program with annotations about the desired layout of the relations.

References

[1] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–154. IEEE Computer Society Press, 2004.

[2] G. Gupta and S. F. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Practical Aspects of Declarative Languages (PADL)*, volume 1753 of *Lecture Notes in Computer Science*, pages 308–323. Springer, 2000.

[3] J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[4] J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. SLAT: Information flow in Security Enhanced Linux, 2003. Available from `http://www.mitre.org/tech/selinux/`.

[5] T. Jaeger, A. Edwards, and X. Zhang. Policy management using access control spaces. *ACM Transactions on Information Systems Security*, Aug. 2003.

[6] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *USENIX Security Symposium*, 2003.

[7] B. Jayaraman and K. Moon. Subset logic programs and their implementation. *J. Log. Program.*, 42(2):71–110, 2000.

[8] S. P. Jones, A. Blackwell, and M. Burnett. A user-centered approach to function in excel. In *Intl. Conf. on Functional Programming*, 2003.

[9] M. Kassoff, L.-M. Zen, A. Garg, and M. Genesereth. PrediCalc: A logical spreadsheet management system. In *31st International Conference on Very Large Databases (VLDB)*, 2005.

[10] A. Kissinger and J. Hale. Lopol: A deductive database approach to policy analysis and rewriting. In *2006 Security Enhanced Linux Symposium*, 2006.

[11] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proc. of 2001 Ottawa Linux Symposium*, 2001. Available from `http://www.nsa.gov/selinx/`.

[12] D. Maier and D. S. Warren. *Computing with Logic: Logic Programming and Prolog*. Benjamin/Cummings Publishers, Menlo Park, CA, 1988. ISBN 0-8053-6681-4, 535 pp.

[13] D. Merrit, J. Paine, and M. Kassof. Special Spreadsheet Issue of AI Expert Newsletter, May 2005. `http://www.ainewsletter.com/newsletters/aix_0505.htm`.

[14] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert. NETRA: Seeing through access control. In *Proc. 4th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 55–66, 2006.

[15] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A logic-based network security analyzer. In *14th Usenix Security Symposium*, 2005.

[16] Quantrix. `http://www.quantrix.com/`.

[17] C. R. Ramakrishnan, I. V. Ramakrishnan, and D. S. Warren. Deductive spreadsheets using tabled logic programming. In *22nd International Conference on Logic Programming (ICLP)*, volume 4079 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, 2006.

[18] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[19] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[20] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at `http://www.cs.sunysb.edu/~stoller/WITS2004.html`.

[21] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag, New York, NY, USA, 1986.

[22] Tresys Technology. Policy tools for Security-Enhanced Linux. Available from `http://www.tresys.com/selinux/`.

[23] Tresys Technology. Security-Enhanced Linux reference policy. Available from `http://oss.tresys.com/projects/refpolicy`.

[24] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer-Verlag, 2005.