ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 1, No. 2, 2006, *pp*. 119-224

Security Downgrading Policies for Practical Software

Jianbo Yao [+] , Jianshi Li

Faculty of Computer Science and Engineering, Guizhou University, Guiyang, Guizhou, 550025, China

(*Received February 26, 2005, accepted April 6, 2006*)

Abstract. Security downgrading policies control information flow and permit information release from a high security level to low security level. Many security downgrading policies are treated as declassification. This paper extend security policies to operations than declassification , the security downgrading policies support downgrading in practical software, each downgrading step is annotated with some operations when some conditions are satisfied. The security type system is formalized as relaxed noninterference.

Keywords: downgrading, security policies, information flow, relaxed noninterference, declassification.

1. Introduction

Language-based information flow security policy is often formalized as noninterference [1][2]; only allow information flow from low security level to high security level. Noninterference is too rigid to use practical software.

Downgrading specifies information flow from a high security place to a low security place, also called confidentiality labels declassification. When a practical software declassifies information properly, there is some reason to accept some information release.

Information secure downgrading through an explicit declassification operation when some primitive conditions are satisfied. Chong and Myers presented security policies for downgrading and a security type system that permits information release where appropriate. The policies are connected to a semantic security condition that generalizes noninterference, and the type system enforced the security condition[3].

Li and Zdancewic formalized downgrading security policies as relaxed noninterference [4]. The decentralized label model(DLM) puts access control information in the security labels to specify the downgrading policy for the annotated data [5]. Robust declassification improves DLM [6][7]. Intransitive noninterference [8][9][10]based on noninterference describe the behavior of systems that need to declassify information. The language $\lambda_{downgrading}$ is a security-typed language[11][12]. Other methods seek to measure or bound the amount of information that is declassified [13][14].

For all security downgrading policies are intension, we therefore propose a security policy framework that supports downgrading in practical software, each downgrading step is annotated with some operations when some conditions are satisfied.

This paper extends Chong and Myers' work that each step in the sequence is annotated with a condition that must be satisfied in order to perform the downgrading [3].

The remainder of the paper is organized as follows. Section 2 presents the motivate example. Section 3 gives the language of security downgrading policies. Section 4 defines a programming language that incorporates security downgrading policies. Section 5 is concludes.

2. Motivating Example

This section gives a motivating example in which data is downgrading. Consider a bid system where each registered bidder submits a single bid to the system. Once all bids are submitted, system opens all bids and the bids compared; the winner is the highest bidder. Before all bids are submitted, each registered bidder may log in the system to examine or amend own bid, but no bidder knows any of the other bids.

[+] Corresponding author. Tel.: +86-851-3627946; fax: +86-851-3627946.
E-mail address: jianbo-yao@21cn.com

The following pseudo-code shows an abstraction of such a system with two bidders, Alice and Bob.

```
1  string{sec ret} password = read_password();
2  string{public} input = read_user_input();
3  string{public} message;
4  if (declassify(password == input)) then message := 'Login OK!'
   else message := 'Login Failed!'
   …;
5  string{sec ret} AliceBid := …;
6  string{sec ret} BobBid := …;
7  string{public} AliceOpenBid := declassify(AliceBid);
8  string{public} BobOpenBid := declassify(BobBid);
   …
```

3. Downgrading Policies

In this section we present downgrading policies which can specify data is declassified though some operation if some condition are satisfied.

Data labeled with a policy $\ell \xrightarrow{c:op} p$ must be treated at security level ℓ, the operator op may be applied to the data provided condition c is true, and the result of the operation is labeled with security policy p.

3.1. Policies

Assuming there is some existing lattice L, such as the decentralized label model [5], and some security policy, such as in [3][4].

Security downgrading policies is presented in Figure1.

$\ell \in L$	Security levels from security lattice L
$p ::=$	Security policies
$\ell \xrightarrow{c:op} p$	Declassification policy
ℓ	Security level policy
$c ::=$	Conditions
d	Primitive conditions
t	True
f	False
$c \wedge c$	Conjunction
$\neg c$	Negation
$op ::=$	Operators
$\lambda x.L$	
$\lambda p.\lambda x.p = x$, $\lambda x.Enc(x)$, $\cdots$,	
$\lambda x.H$	

Figure 1. Security downgrading policies

Here conditions are used to express when it is appropriate to declassify data; operator express declassification operation after some conditions are satisfied.

ℓ is a security level, but ℓ is a security policy for declassification.

Operator op is defined a $\lambda-$ calculus, non-empty set of operation function. The operation functions have the order [4]:

$$\lambda x.L \sqsubseteq \lambda p.\lambda x.p = x \sqsubseteq \lambda x.H$$

Those operation functions operate on any data to change security level of the data, but not to change

value of the data. That is:

$$(\lambda x.L)\, data_H \rightarrow data_L$$
$$(\lambda x.L)\, data_L \rightarrow data_L$$
$$(\lambda p.\lambda x.p = x)\, password \rightarrow (\lambda x.password = x)$$
$$(\lambda x.H)\, data_H \rightarrow data_H$$
$$(\lambda x.H)\, data_L \rightarrow data_H$$

Define an ordering $\leq$ on policies:

$$\frac{\ell \,\hat{\mathrm{o}}_L\, \ell' \qquad p \leq p', c' \Rightarrow c, op \,\hat{\mathrm{o}}\, op'}{\ell \xrightarrow{c:op} p \leq \ell' \xrightarrow{c':op'} p'}, \qquad \frac{\ell \,\hat{\mathrm{o}}_L\, \ell'}{\underline{\ell} \leq \underline{\ell'}},$$

$$\frac{}{\underline{\ell} \leq \ell \xrightarrow{t:op} \underline{\ell}}, \qquad \frac{}{\ell \xrightarrow{t:op} \underline{\ell} \leq \underline{\ell}}$$

The relation $\leq$ is not a partial order, as it is not anti-symmetric.

If there is the equivalence relation $\equiv$ over operators op such that $op \equiv op'$, then our framework reduce to Chong and Myers' framework [3]; If there is the equivalence relation $\equiv$ over conditions c, then our framework reduce to Li and Zdancewic's framework [4].

3.2. Review Motivating Example

In motivating example, we can use the security policy $H \xrightarrow{t:op} \underline{L}$ for password-checking, op is $\lambda p.\lambda x.p = x$ and the primitive condition is permanent true, then $password$ is downgrading through $(\lambda p.\lambda x.p = x)\, password \rightarrow (\lambda x.password = x)$; we can also use the security policy $H \xrightarrow{c:op} \underline{L}$ for open bids, Primitive condition c is true if and only if both Alice and Bob have submitted their bids; op is $\lambda x.L$, then for c is true, AliceBid or BobBid are downgrading through $(\lambda x.L)\, AliceBid_H \rightarrow AliceBid_L$ and $(\lambda x.L)\, BobBid_H \rightarrow BobBid_L$.

4. A Language for Local Downgrading

In this section we present a programming language $\lambda_{downgrading}$, based on the security-typed $\lambda - calculus$, that supports downgrading.

4.1. The Language

The language syntax is presented in Figure 2. Compared [3] language, we reduce explicit declassification operator.

$v ::=$		Values
	x	Variables
	n	Integers
	$()$	Unit
	$\lambda x:\tau.[p]e$	Abstraction
	m^{τ}	Memory locations
$e ::=$		Expressions
	v	Values
	ee	Application
	$\mathrm{ref}^{\tau} e$	Allocation
	$!e$	Dereference
	$e := e$	Assignment
	$e;e$	Sequence

$\beta ::=$	Base types
int	Integers
unit	Unit
$\tau \xrightarrow{p} \tau'$	Functions
τ ref	References
$\tau ::=$	Security types
β_p	Base types with policies

Figure 2. Syntax of the language $\lambda_{downgrading}$

4.2. The Type System

$\tau \prec: \tau'$ denotes that τ is a subtype of τ'. The subtyping rules are listed Figure 3 and typing rules are in Figure 4.

$$\frac{\beta \prec: \beta' \quad p \le p'}{\beta_p \prec: \beta'_{p'}}, \qquad \frac{}{\beta \prec: \beta}, \qquad \frac{\beta \prec: \beta' \quad \beta' \prec: \beta''}{\beta \prec: \beta''}, \qquad \frac{p' \le p \quad \tau_1' \prec: \tau_1; \tau_2 \prec: \tau_2'}{\tau_1 \xrightarrow{p} \tau_2 \prec: \tau_1' \xrightarrow{p'} \tau_2'}$$

Figure 3. subtyping rules

$$T-Var \quad \frac{\Gamma(x) = \tau}{pc, \Gamma \vdash x : \tau}$$

$$T-Int \quad \frac{}{pc, \Gamma \vdash n : \mathrm{int}_p}$$

$$T-Unit \quad \frac{}{pc, \Gamma \vdash () : unit_p}$$

$$T-Loc \quad \frac{}{pc, \Gamma \vdash m^{\tau} : \tau \ \mathrm{ref}_p}$$

$$T-Sub \quad \frac{pc, \Gamma \vdash e : \beta_p \ \mathrm{ref}_{p'}}{pc, \Gamma \vdash !e : \beta_{p \cup p'}}$$

$$T-Deref \quad \frac{pc, \Gamma \vdash e : \beta_p \ \mathrm{ref}_{p'}}{pc, \Gamma \vdash !e : \beta_{p \cup p'}}$$

$$T-Seq \quad \frac{pc, \Gamma \vdash e_1 : unit_p; pc, \Gamma \vdash e_2 : \tau}{pc, \Gamma \vdash e_1; e_2 : \tau}$$

$$T-Abs \qquad \frac{pc,\Gamma[x\mapsto\tau]\left|-e:\tau'\right.}{pc,\Gamma\left|-\lambda x.\tau.[p]e:\left(\tau\xrightarrow{p}\tau'\right)_{p'}\right.}$$

$$T-App \qquad \frac{pc,\Gamma\left|-e_1:\left(\tau\xrightarrow{pc'}\beta'_{p'}\right)_p\right.;pc,\Gamma\left|-e_2:\tau;pc\le pc'\right.}{pc,\Gamma\left|-e_1e_2:\beta'_{p\cup p'}\right.}$$

$$T-Alloc \qquad \frac{pc,\Gamma\left|-e:\beta_p\right.;pc\le p}{pc,\Gamma\left|-\mathrm{ref}^{\beta_p}e:\left(\beta_p\ \mathrm{ref}\right)_{p'}\right.}$$

$$T-Assign \qquad \frac{pc,\Gamma\left|-e_1:\beta_p\ \mathrm{ref}_{p'}\right.;pc,\Gamma\left|-e_2:\beta_p\right.;pc\cup p'\le p}{pc,\Gamma\left|-e_1=e_2:unit_{p'}\right.}$$

$$T-Mem \qquad \frac{\forall m^\tau\in dom(M).\mathrm{T},\emptyset\left|-M\left(m^\tau\right):\tau\right.}{\left|-M\right.}$$

Figure 4. typing rules

Definition 4.2.1. $\Re(e)$ erases all security level label in e and returns a simply-typed $\lambda-term$.

Theorem 4.2.1. Relaxed Noninterference

if $\left|-e:\beta_{\underline{L}}\right.$, *then* $\Re(e)\equiv f\left(if\ c_1\ then\ op_1\ \mathrm{data}_{\mathrm{H}}1\right)\ldots\left(if\ c_1\ then\ op_k\ \mathrm{data}_{\mathrm{H}}k\right)$, where $\forall i, data_H i\notin FV(f)$.

proof ,

By induction on all $\Gamma\left|-v:\beta_{\underline{L}}\right.$ and $\Gamma\left|-e:\beta_{\underline{L}}\right.$.

This theorem shows that a type-safe can only leak secret information in controlled ways.

5. Conclusion

We have presented framework for declassification security policies, and incorporated the security policies in a security type system. The framework extends the security policies to some operations than declassification increases expressiveness of the security policies.

In the language setting of a security type system, these downgrading policies are connected to some operations operator when primitive conditions are satisfied. Data labeled with a policy $\ell\xrightarrow{c:op}p$ is treated at security level ℓ, the operator op may be applied to the data provided condition c is true, and the result of the operation is labeled with security policy p.

The security policies are enforced to control information flow in security type system for practical software.

Our security type system is formalized as relaxed noninterference.

6. References

[1] A. Sabelfeld and A. C. Myers, Language-based information-flow security. IEEE J. Selected Areas in Communication, 21(2003)1.

[2] John Mclean, Security models and Information Flow, In Proc. IEEE Symposium on Security and Privacy, IEEE

Computer Society Press, 1990, pp 180-187.

[3] Stephen Chong and A. C. Myers, Security Policies for Downgrading, CCS'04, Washington, DC, USA, October 25-29, 2004.

[4] Peng Li and Steve Zdancewic, Downgrading Policies and Relaxed Noninterference, POPL'05, Long Beach, California, USA, January 12-14, 2005.

[5] A. C. Myers and B.Liskov, Complete, safe information flow with decentralized labels, In Proc. IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 1998, pp 186-197.

[6] Steve Zdancewic and A. C. Myers, Robust declassification, In Proc. IEEE Computer Security Foundations Workshop, Cape Breton, Canada, IEEE Computer Society Press, June 2001.

[7] A. C. Myers, A.Sabelfeld and Steve Zdancewic, Enforcing Robust declassification, In Proc. IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, June 2004, pp 172-186.

[8] S. Pinsky, Absorbing covers and intransitive non-interference, In Proc. IEEE Symposium on Security and Privacy, 1995, pp 102-113.

[9] A. W. Roscoe and M. H. Goldsmith, What is intransitive noninterference? In Proc. 12th IEEE Computer Security Foundations, Workshop, 1999.

[10] J. Rushby, Noninterference, transitivity and channel-control security policies, Technical Report CSL-92-02, SRI, Dec. 1992.

[11] A. Banerjee and D. A. Naumann, Secure information flow and pointer confinement in a Java-like language, In Proceedings 15th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada, June 2002, pp 253-267.

[12] A. C. Myers, Flow: Practical mostly-static information flow control, In Proc. 26th ACM Symp. POPL, 1999, pp 228-241.

[13] A. D. Pierro, C. Hankin and H. Wiklicky, Approximate non-interference, In Proc. 15th IEEE Computer Security Foundations, June 2002, pp 1-15.

[14] G. Lowe, Quantifying information flow, In Proc. 15th IEEE Computer Security Foundations Workshop, June 2002.