Replication does survive information warfare attacks*

J. McDermott
Naval Research Laboratory
Washington, DC 20375-5537, USA
mcdermott@itd.nrl.navy.mil

Abstract

Recent literature on information warfare has suggested that general replication is not useful in dealing with storage jamming attacks. We show that special cases of replication are useful not only in detecting but also in recovering from storage jamming attacks.

Keywords

Information warfare, storage jamming, unauthorized modification, Trojan horse

1 INTRODUCTION

Ammann, Jajodia, McCollum, and Blaustein define *information warfare* as the introduction of incorrect data intended to hinder the operation of applications that depend on the database (Ammann, Jajodia, McCollum, and Blaustein, 1997). In describing their approach to surviving these kinds of attacks on databases, imply that replication is not useful in dealing with information warfare attacks. In this paper we present results to the contrary, i.e. replication can be used (carefully) to both detect and survive information warfare attacks, on a practical basis.

McDermott and Goldschlag (McDermott and Goldschalg, 1996a), (McDermott and Goldschlag, 1996b) define *storage jamming* as "malicious but surreptitious modification of stored data, to reduce its quality. The person initiating the storage jamming does not receive any direct benefit. Instead, the goal is more indirect, such as deteriorating the position of a competitor." This is essentially the same as information warfare, and we adopt the latter term. To provide context, Amman et al. specifically do not consider Trojan horses within the database system (called *internal jammers* (McDermott and Goldschalg, 1996b)), but instead consider a wide range of attacks other than Trojan horses. Both groups agree that Trojan

* This work was supported by ONR. Any opinions, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views, policies, or decisions of the Office of Naval Research or the Department of Defense.

Database Security XI T.Y. Lin and Shelly Qian (Eds)

horses are more effective attackers, since they can access data which the human attacker cannot. McDermott et al. show how to detect sophisticated attacks by Trojan horses inside the database system but do not address recovery or continued operation. Amman et al. do not address detection. Instead, they show not only how to assess damage after an attack, but also how to continue operation with partially damaged data. Here, we show how replication can be used to not only detect attacks, but to assess damage and continue operation, thus surviving information warfare attacks.

We borrow some terms from Ammann et al. and refer to data damaged by an attack as either *red data* or *off-red data. Red* data is unsafe to use; *off-red* has been damaged, but may be used. *Green data* is valid and has not been damaged. We also use *red* and *green* to describe values that are to be stored as data, by the attacker and the defender respectively.

2 REPLICATION AS A DEFENSE: DETECTION

Replication in general is problematic in an information warfare context. Under many replication approaches, *red* data can be replicated automatically and precisely to many locations. However replication works as a defense if we use *(one-copy serializable) logical replication over distinct database systems.* Many replication algorithms copy data values from the source data item to its replicas. However, logical replication copies the command that caused the source data item to change. The command is executed at each replica's site and, because of one-copy serializability, results in the same new value for the replica. If we assume a distinct provenance (defined in the next section) for the database system software at each site, then the Trojan horse will not be replicated at all sites. An attack must compromise multiple, possibly heterogeneous, host programs, an unlikely event in practical systems. Even if the attackers can succeed at every site, the attack still may fail. If the Trojan horses are not able to deliberately malfunction in a one-copy serializable fashion, their *red* values will diverge. This can be ensured by restricting communication between the sites to just the protocols needed to carry out the authorized replication. So we can expect a scheme using n replicas to detect up to n-1 cooperating Trojan horses and possibly detect an n-Trojan horse attack.

Detection is simple in the replication defense. There is a detection process at each source or replica site. Following changes to protected data, the process at the source site computes a checksum over the changed data and sends it to each replica site, along with the identification of the change. After the logical update is performed at a replica site, the detection process at the replica site computes its own checksum and compares it to the checksum transmitted by the source site detection process. If there is disagreement, there is a problem. Checksums are not essential to the approach and are merely used to facilitate efficient comparison. The granularity of the comparisons or checks is a tradeoff between speed and storage. Comparisons over individual data items allow quicker response to attacks but take more storage to perform. We also do not need to check every change, since the insertion of bogus data at some sites will ultimately diverge the copies.

If we establish checksums over our entire database, detection can be effective against both external jammers and internal jammers. (External jammers attack files outside their host application, e.g. a Trojan horse hosted by an Oracle database system that attacks Mathematica files is an external jammer.) Indistinguishability (McDermott and Goldschlag, 1996b) comes for free, without our being able to define or verify it.

Table 1 . The *refueling* relation

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 1	12 000
Sword 2	McGuire	Coke 2	12 000
Axe 1	O'Hare	Coke 2	33 000

Table 2. The *tanker* relation

Tanker	Fuel Available
Coke 1	33 000
Coke 2	19 000

We use the aircraft refueling example of Ammann et al. Suppose we have a relational database with two relations *refueling* and *tanker*. Both relations are replicated at three sites: *cactus*, *yucca*, and *sorrel*. The checksums for *refueling* and *tanker* are r_1 and t_1, respectively. At site *yucca*, a command is issued to "**update** *refueling* **set** *tanker* = 'Coke 2' **where** *aircraft* = 'Sword 1'", but there is a Trojan horse in the database system at *yucca* that sets *tanker* = 'Coke 1' where *aircraft* = 'Sword 2'. The new (incorrect) checksum for refueling is r_2, which is sent to sites *cactus* and *sorrel*, along with the command "**update** *refueling* **set** *tanker* = 'Coke 2' **where** *aircraft* = 'Sword 1'." At both *cactus* and *sorrel*, the requested change is made to *refueling*, but the detection processes compute a different checksum r_3, because the result of correctly executing the command is different. Either detection process can now report a problem because $r_2 \neq r_3$. Notice that, in this example, we must compute checksums for relation *tanker*, because the Trojan horse may have modified the *tanker* relation while performing "**update** *refueling* **set** *tanker* = 'Coke 2' **where** *aircraft* = 'Sword 1'" correctly. If the Trojan horse had been at site *cactus* the attack would still be detected by the difference in checksums. (We defer an example of damage assessment and continued operation until later in the paper.)

It is important that the detection process be separate from the database system. Otherwise, it might be possible for the Trojan horse to send checksums for *green* values to other sites while writing *red* values at its own site. Furthermore, the detection process must be trusted, i.e. it must be high assurance software that is protected from tampering. Finally, we are assuming that the Trojan horse can be located and removed using existing system administration tools.

Although this approach is reminiscent of Byzantine generals approaches, we do not recommend extending it to carry out a similar automatic agreement protocol. The foremost reason for this is that our approach is intended to work with shrink-wrapped general purpose software. It is unlikely that software vendors will modify their products to carry out the cryptographically protected voting protocols needed

to reach Byzantine agreement. A less important reason is that use of such protocols for every update would seriously impact the performance of most database systems. At present it is more expedient to detect the attacks and then remove the Trojan horse.

2.1 Distinct Provenance

Software that is created, delivered, installed, and maintained by distinct sets of people has a distinct provenance. Distinction can be forced to many levels by a variety of techniques. Since multidatabase techniques allow replication over heterogeneous systems, the database systems at each site can be different, even if they are shrink-wrapped general purpose software packages.[1] Shrink-wrapped general purpose software packages (e.g. the database system software) can be purchased through blind buys, which simulates distinct provenance. Applications, site-specific software, macros, etc. can be developed using clean room techniques. In a clean room approach, developers provide inspected source code to each site. The source code is converted to executable form (e.g. compiled and linked, converted to p-code) and installed at the operational sites by personnel distinct to each site. Maintenance and administration can likewise be separated site-wise by clean room techniques. Our notion of distinct provenance is not the same as *n*-version programming. We are not trying to tolerate inadvertent bugs but to deny an attacker access to multiple sites. The expectation is that we have now forced would-be attackers to compromise multiple host programs in very sophisticated ways. A practical *n*-Trojan horse attack can only succeed if all *n* Trojan horses can maintain one-copy serializability over all changes to their red data and internal states. Since the successful Trojan horses cannot be replicas of each other, this is problematic for the attacker. If we assume, say a software development team, has *m* members who understand the software[2] then the *n*-Trojan horse attack reduces to an *mn*-person manual attack.

In theory, a distinct provenance is possible. In practice, some software may have commonality. Some distinct software will either have been developed with the same tools or be based on the same packages. This raises the question of Trojan-horse-writing Trojan horses (McDermott, 1988). Fortunately, a would-be attacker introducing an attack via widely-used software faces a significant problem. The problem is that the Trojan horse's lifetime is now likely to be expended against systems other than the target. The Trojan horse will trigger on sites that are not the intended target. The attacker must now arrange to turn off the Trojan horse in systems that are not targets or risk premature discovery of the attack. Attacks via automatic data input systems face the same problem. More red data must be created, and not all of it will be put in the target database. This increases the chance of someone detecting the attack by inspection of the data.

2.2 Manual Attacks

---

[1] Introduction of heterogeneity may require the use of trusted mapping functions that are assured to map the logical update commands in a way that preserves checksums.

[2] The point here is that *m* is not as large as the entire team, but in a well-managed properly assured software development program greater than unity. Under poorly-managed, low-assurance development, we are not sure any defense is possible.

Logical replication is clearly a problem for Trojan-horse-based attacks because those attacks function by "disobeying" the commands given to the software. So we have frustrated the most effective means of attack. But what about less effective manual attacks? Manual attacks are carried out by giving malicious commands to the database system. We can deal with manual attacks in one of two ways: 1) by incorporating an *n*-person rule, or 2) by incorporating transaction control expressions (Sandhu, 1990). An *n*-person rule requires *n* humans outside the system to agree to a change to the database. Transaction control expressions are a more general form of this concept. They require multiple users to agree to specific conditions defined on specific steps of a transaction. In either case, we assume that data manipulation commands are legitimate unless all *n* persons can collude.

We also note that in newer automated systems, the amount of manual input to a database is less than in the past. For example, tanker aircraft may have on-board software that automatically reports the amount of fuel carried. Refueling assignments to aircraft may be calculated by a decision aid program. This appears to be just moving the problem around, but in fact it reduces the opportunities for manual attacks. Information warfare attacks via automatic data input suffer from the same weakness as Trojan horses written into mass-produced software (see below).

2.3 Application Attacks, Interface Attacks, etc.

Careful readers will question whether application programs can be abused to simulate the advantages of manual attacks while avoiding transaction control expressions or *n*-person rules. If an application outside the database contains the attacking Trojan horse, it can submit commands to insert bogus values and the database system will replicate the bogus commands as though they were manual commands. Fortunately, this type of attack is frustrated by replicating the application software, i.e. defensive replication is not limited to database systems. The same type of attack can be made via any software (we hope not via hardware or firmware!) that lies between a system's input devices and its output devices. Careful replication of these components will suffice to detect such attacks just as the basic database attacks are detected. Our approach does have trouble with the connections between a system and its I/O peripherals. When we finally reach the devices that lie at the boundaries of our system, things become unclear. In a theoretical sense, we can define the problem away by saying that attacks that modify data as it is being put in or out are not information warfare attacks. In a practical sense, we would have to limit our replication to components that handle the most critical data.

3 REPLICATION AS A DEFENSE: DAMAGE ASSESSMENT

Ammann et al. introduced the important concept of *damage markings*. Damage markings are attributes that indicate the degree of damage that has been assessed upon a particular data item. We also adopt damage markings and use their scheme.

Leaving other considerations such as system errors aside, when a check fails and we detect an attack, we should expect that either the source database system has been compromised or the replicas that failed the check have been compromised. All systems participating in the defense should be alerted. We assume that database administrators and support teams will eventually locate the Trojan horse and remove it. Data items relating to the change that failed the check should all be marked *red*, even though some will in fact be *green*. The correct values can be determined by manual inspection, by simple majority vote over all copies of a data

item. Correct copies of the data are then marked *green*. Copier transactions can use the *green* replicas to repair damaged copies of data items.

Suppose we look at damage assessment in our refueling example. *Red* markings on the copies of *refueling* at sites *cactus* and *sorrel* can be changed to *green* by the damage assessment transaction. Note that markings for relation *tanker* need not be changed at any point during this attack. When the detection processes at sites *cactus* and *sorrel* detect the attack, all tuples of the *refueling* relation can be temporarily marked *red*, on the basis of the text of the command that failed the checksum. Damage assessment in this case can be accomplished by majority vote, which allows us to identify the second tuple of *yucca*'s copy of *refueling* to be damaged. Tables 3 and 4 indicate the state of the database after damage assessment, with *red* data underlined.

Table 3. Marking the damaged *refueling* relation at site *yucca*

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 2	12 000
Sword 2	McGuire	Coke 1	12 000
Axe 1	O'Hare	Coke 2	33 000

Table 3 . Marking the damaged *refueling* relation at sites *cactus* and *sorrel*

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 2	12 000
Sword 2	McGuire	Coke 2	12 000
Axe 1	O'Hare	Coke 2	33 000

4 REPLICATION AS A DEFENSE: CONTINUED OPERATION

The use of logical replication may allow us to disconnect compromised systems until the Trojan horse can be disabled. If an uncompromised site can act as a source site, it can take over from a compromised source. Replica sites that do not originate data are also easily disconnected.

A more complex approach would logically "disconnect" compromised data items (e.g. classes or relations)

4.1 Defensive Partition

If the compromised site is not the source of the data or there is an alternate source site, then the replicated database system can be partitioned into a damaged and an undamaged component. The partition can take place after damage assessment and could be decided on the basis of an agreement algorithm, just like the damage as-

sessment. Any compromised sites are placed in the damaged component. The sites in the undamaged component can continue to operate normally. Sites in the damaged component would only be allowed to submit read requests to sites in the undamaged partition.

4.2 Single-Source Data

If the replication is done with only one source site, and that site is compromised, then we conjecture that we can still use a modified version of the continued operation protocol of Ammann et al. Their protocol uses transactions that distinguish between inputs, outputs, pure reads, updates (read and write), and blind writes, as well as insert and delete. Our modifications for continued operation under single-source replication are:

1. We do not use the *off-green* marking. Correct[3] values of every data item will be available for repair of every detected attack. We decide at database design time whether a data item will be marked *red* or *off-red* during damage assessment.
2. We do not use the Coincidental Damage Deletion rule because it may allow incorrect deletion of *off-red* data that will ultimately be marked *green* by a damage assessment algorithm[4]. We do use the other rules listed below. Notice that the remaining rules have been modified to incorporate replication.
    a. Confinement: A normal transaction T that attempts to read, update, blind write, or delete a data item accesses any available *green* copy. If no *green* copy is available, the normal transaction attempts to read an *off-red* copy. If no *off-red* copy is available, then T rolls back. A normal transaction may not create *red* data.
    b. Propagation: If a transaction T reads data marked *off-red*, then any output by T is marked *off-red*. Transaction T may not update, blindly write, or delete data for which a *green* copy is available. Transaction T may not create *off-red* data.
    c. Coincidental Repair of Off-Red Data: If a transaction reads only *green* data then any *off-red* data item it writes blindly is marked *green*.
3. We simplify the definition of consistency by leaving out the acceptable but not necessarily consistent integrity constraints, giving us the following definition of integrity
    a. For each integrity constraint $i \in I$, where I references exclusively *green* data, i holds.
    b. For each integrity constraint $i \in I$, where I references data items $x_1, \ldots x_n$ that are not *green*, there exist values for $x_1, \ldots x_n$ such that i is satisfiable.

---

[3] but not necessarily up-to-date

[4] We assume that the presence of correct copies of the data makes it likely that this repair will take place in a relatively short period of time.

4. All data is initially marked *green*. Markings are changed by a damage assessment algorithm, from *green* to *red* or *off-red*, *iff* the data is damaged. Damage assessment transactions do not change any data, but correctly identify valid copies of data items that have damaged replicas. Markings are changed by a copier transaction repairing damage, from *red* or *off-red* to *green*.

A normal (i.e. not a copier, attacker, or damage assessor) transaction *T preserves consistency* if, given a consistent and all *green* state S_1, *T* produces a consistent all *green* state S_2.

We now pose a theorem analogous to one of Ammann et al., namely

4.3 Theorem

Suppose a consistency preserving normal transaction *T* follows the modified continued operation protocol defined above, and S_1 is a consistent state of the possibly damaged database. Then state S_2, the state resulting from the application of *T* to S_1, is consistent.

Proof:

The Confinement rule prohibits transaction *T* from accessing any *red* data. Transaction *T* cannot violate *I* by reading or writing *red* data.

1. The Propagation rule allows *T* to cause *green* data to become *off-red*. Consider an integrity constraint *i*. If some copy of a data item *x* referenced in *i* is green in state S_1 and becomes off-red in state S_2 as a result of transaction *T*'s actions, then there must be values for a copy of data item *x* that satisfy *i* , that is value of *x* in S_1. (Even though green replicas of *x* are often available, all copies of *x* may be temporarily marked *red* or *off-red* by a damage assessment transaction.) A transaction that reads *off-red* data under the Propagation rule cannot modify *green* data without causing it to become off-red. For this reason, *green* data in the new state S_2 must also have been marked *green* in state S_1. Integrity constraints in *I* are satisfied in S_2.
2. The Coincidental Repair of Off-Red Data rule allows a normal transaction *T* to change the marking of an *off-red* data item *x* to *green* when writing blindly, if *T* only reads *green* data. Since transaction *T* is consistency preserving, the values it writes when changing *x* satisfy *I* in S_2.

To return to our running example: for continued operation we could at first not use any data from relation *refueling*. After damage assessment, the relations would be marked as shown by Tables 3 and 4. We could read and modify the copies of *refueling* at sites *cactus* and *sorrel* even though the damage was not repaired. Ultimately, a copier transaction could repair the "Sword 2" tuple of *yucca*'s copy of refueling, by copying correct values from either *cactus* or *sorrel*.

4.4 Stored Procedures

Stored procedures are widely used in current databases. Their impact on storage jamming is problematic. First of all, the stored procedure mechanism is an ideal tool for building efficient, sophisticated jammers. Stored procedures also make good hiding places. On the other hand, all but the most sophisticated jamming attacks against stored procedures are probably too risky for the attacker. Plausible

values for passive data items are easy to generate, either by arithmetic or by copying components (e.g. fields). Applying simple arithmetic to the text of a stored procedure does not necessarily result in a plausible, valid program text. Copying substrings of a program text into the target procedure may result in a valid program, but probably not a plausible one.

Predictability is also an issue for the attacker. The modified procedure may exhibit spectacular behavior that immediately reveals the Trojan horse. Programs that can automatically generate valid program texts that also implement specific algorithms are still in the research stage. They are also relatively large, i.e. on the order of general purpose database system software, so they would be difficult for the attacker to hide. Inserting bogus code into multiple stored procedures could result in a combinatorial explosion of bad data that would also reveal the attack. It is possible that future research in automatic code generation could make it possible to build a small malicious program that surreptitiously modifies stored procedures.

Stored procedures require extra care on the part of the defenders. They must be replicated, but with distinct provenance. They should not be automatically copied or translated to the various sites, but should be reviewed outside the database systems and then installed manually. If distinct provenance is maintained, replication should be an effective means of defending against jamming of stored procedures.

5 CONCLUSIONS

Before presenting our conclusions, we would like to discuss some key assumptions we are making, so the that the application of our results will be clear:

- We assume that some malicious software can be introduced into most systems during their lifetime. We assume that introducing specific malicious software into multiple sites is problematic and cannot be done repeatedly or at will.
- We assume malicious software or users can be removed from a system soon after they are detected. This is not always so in real life, but it is possible in systems following best practice.
- The following software components must be trusted: the detection process that computes, compares, and transmits checksums, any mapping functions used to translate logical updates to site-specific languages, damage assessment voting or agreement algorithms, and copier transactions used to repair damage. To warrant this trust they must be correct, unbypassable, and tamper-proof. We assume sufficient access control, audit, and cryptographic systems to make this be so.

Replication via logical updates is a viable defense that allows detection of, damage assessment after and continued operation during information warfare attacks. With n replicas, logical replication is effective in detecting automatic (Trojan horse) attacks involving less than mn person collusion, where m is the number of members of a software development or maintenance team. With n-person data entry, logical replication is effective in detecting manual attacks involving less than n-person collusion. With transaction control expressions, the likelihood of successful manual attack is even less.

Detection of an attack by logical replication results in an undamaged copy of the target data, at either the source or the replica site. A simple majority of undamaged copies is sufficient to identify the correct values. Even if there is no majority, pos-

session of the text of the offending command will allow (admittedly tedious) identification of the correct value.

The continued operation protocol of Amman, Jajodia, McCollum, and Blaustein can be used to operate a replicated database system prior to identification of the correct values. Once the correct values have been identified, the database can operate from either *green* copies or our modification of the original continued operation protocol. The existence of identifiably correct copies makes it possible to intentionally partition the damaged database system, thus isolating the offending subsystem.

At present we are prototyping proof-of-concept software for a replicated architecture defense. Our target system is SQL Server running on Windows NT. Future work should include more sophisticated continued operation protocols that account for both communication failures and site failures. Specific damage assessment algorithms, accompanied by improved damage marking schemes may be beneficial to improved recovery from attacks.

5.1 Acknowledgements

We would like to thank David Goldschlag, Carl Landwehr, and Robert Gelinas for their contributions to this paper. The thoughtful comments of the anonymous referees have improved the paper considerably.

6 REFERENCES

Ammann, P. Jajodia, S., McCollum, C., and Blaustein, B. Surviving information warfare attacks on databases. In *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, Oakland, California, May, 1997.

McDermott, J. A technique for removing an important class of Trojan horses from high-order languages. In *Proceedings 11th National Computer Security Conference*, Baltimore, 1988.

McDermott, J. and Goldschlag, D. Storage jamming. In *Database Security IX: Status and Prospects* (D.Spooner, S. Demurjian, and J. Dobson, eds.), Chapman and Hall, London, 1996.

McDermott, J. and Goldschlag, D. Towards a model of storage jamming. In *Proceedings IEEE Computer Security Foundations Workshop*, Kenmare, Ireland, June 1996.

McDermott, J. Practical defenses against storage jamming. 20th National Information Systems Security Conference, Baltimore, MD, October 1997.

Sandhu, R. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects* (J. Jajodia and C. Landwehr, eds.), North-Holland, 1990.

7 BIOGRAPHY

John McDermott has been active in computer security research since 1987. He received his Ph. D. from George Mason University in 1994. His current interests are data integrity and security for mobile agents.