Virtual Machines + "Reincarnation" attacks ⇒ Potent Malicious Middleware

Dhananjay S Phatk {phatak@umbc.edu}
UMBC, CSEE Dept

John Krautheim, Mike Oehler
National Security Agency, Fort Mead, MD

Bruce Howell, Alan Sherman John Pinkston and Chintan Patel
UMBC CSEE Dept

Abstract

We demonstrate that virtual machines enable new more potent forms of Malicious Middleware that make enforcing licenses and DRMs a lot harder. To this end we first unveil the "Re-incarnation" attack which is peculiar blend of the Man-in-the-Middle and replay attacks. We then illustrate examples of licences that were easily circumvented without touching the application code (i.e., without any need to break encryption schemes that could be deployed by the application or reverse engineering the application code.

This leads to fundamental issues regarding software-only DRM enforcement. Moreover, we demonstrate that a trusted hardware and trusted boot to bring the system into a known state followed by an attestation (i.e., certification of the state by an external/third party entity) are still not sufficient to guarantee DRM protections. Our results indicate that a lot more intricate hardware and software infrastructure is needed to guarantee full DRM protection.

I Introduction

VMware, Parallels and other vendors have been continually evolving the Virtual Machine concept since the mid/late 80s [9]. The main goal was ease of hardware and software integration, upgrading and maintenance and efficient utilization of hardware (for example the idea of a Virtual Memory was originally introduced in order to accommodate programs bigger than the size of the physical RAM (or magnetic core) memories available at that time).

Virtual Machines have significant security implications [2, 3, 4, 7, 8]. They could be used to enhance security [5, 6]. Virtual Machines could also be used to implement "malware", i.e., malicious software [1]. Likewise vulnerabilities specific to virtual machines can be exploited to mount attacks specialized against them [7].

In this paper we demonstrate new methods to defeat licensing and software protection mechanisms without breaking the application level encryption or reverse-engineering software. To this end we first illustrate the "Re-incarnation" attack. This mode of attack brings out the fundamental limits of software-only DRM and license/access control mechanisms. We then demonstrate actual licences (full industrial grade commercial licenses) that we could easily circumvent in our laboratory using the "Reincarnation" attack.

We then show that trusted hardware together with a trusted boot into a known state followed by attestation (i.e., an independent, third-party verification of the "state") which is the current state-of-the-art, still cannot guarantee proper DRM/license enforcement. Such a sequence of safeguarding procedures (starting with a trusted platform module that guarantees the integrity of the hardware, a secure boot that guarantees that what gets booted is only a trusted, verified copy of the Operating System, followed by an attestation) is no dobut a very effective security paradigm. It can be a good safeguard against many vulnerabilities (such as boot-sector trojans etc). However when it comes to DRM and license protection, this security paradigm is not sufficient to provide unconditional, analytically provable guarantees regarding the encofcement of DRM policies.

We then discuss possible approaches to tackle this issue of software protection. Our analysis indicates that a lot more elaborate hardware and software infrastructure is needed to guarantee that software that falls in the hands of a remote client cannot be reverse-engineered or be re-used in any manner not consistent with a predetermined and mutually agreed upon manner.

We would like to clarify that our goal is not to champion and/or further the goals of the software industry. Rather the later part of this paper includes a theoretical investigation of the fundamental limits of software-only mechanisms to protect software. Note that such strong guarantees regarding the protected use of software are needed in in Military/Defense scenarios. Sensitive equipment/enties (for instance a reconissance plane/drone or a robot-soldier of the future) could be provided to "allies". Should the events take a different turn and the should the erstwhile allies become advresaries, they should not be able to reverse engineer a single bit of the control software, neither should they be able to use the entites (reconissence drones, robot cops ....) without the permission of the original vendor (otherwise the same drones and robot cops could be used against the supplier !).

I.1 Background

Simple checks based on times-tamps can be defeated. For example if the software merely checks for current time (many software products typically have a 30 day trial period) it is trivial to set back the BIOS clock of the virtual machine and turn off network communications (to prevent access to external time servers) and run the software. Simply setting BIOS clock is not the novelty in this work.

I.2 Identifying and defeating clock manipulations

Note that an easy way to safeguard against simple clock resettings is to maintain **additional (state) information**. For instance the application would maintain logs of how many times it was invoked, how much time elapsed between each invocation, etc. It could get even more elaborate and create a nonce based on the system parameters (including processor serial number, BIOS revision number, release date, network card's MAC address and a large number of other values related to all peripherals such as hard drives, displays, removable media drives etc). That nonce could be used to derive a key to encrypt the number of times the application got used, invocation times etc.

Since it is hard to keep track of which parameters might get used in a nonce, the simple strategy of maintaining additional state information was very effective until virtual machines appeared....

I.3 Virtual Machines Enable State Rollback

The application may gather all the information it wants, it could even randomize the information it gathers, (for instance it could track the keystrokes input by different users, and use the mouse motion patterns, timings etc. to generate nonecs that are unique to each user's invocation and usage of the application).

The question is where can the application keep this state information ? certainly keeping it inside the virtual machine is no good because the adversary could simply discard the copy of the machine after a data processing session and re-start from a pristine copy every single time as summarized by the procedure below.

Case I : Local State information only This category includes licenses that maintain local state information (including encryption of usage parameters etc). The attribute that distinguishes this category from the others is that an end-to-end encrypted challenge-response authentication with some "license/authentication/activation server" entity is not needed to "unlock the software".

Such local-state based safeguards can be easily circumvented as indicated below :

(1) Install a copy of the application on a virtual machine. This copy is henceforth called the Pristine-Virtual-Machine (PVM) that includes the freshly installed application).
(2) Every time that software is to be used, copy the PVM and call the copy the Temporary-Virtual-Machine (TVM)
(3) The virtual-BIOS-clock of the TVM is always initialized to a suitable time AFTER the software got installed (most software programs check that system time is past the value snapshotted when the application got installed). Note that the TVM is simply a FILE. so it can always be started in the exact same state (by controlling all the inputs at boot time, it is possible to make sure that the random number generators end everything comes up in exactly the same pristine state so that the application found itself in when it got installed in invoked the first time).
(4) The TVM's Internet access is heavily filtered: it has a private Internet address (for example 192.168.x.y) and lists the Virtual Machine running the host operating system as the gateway.

The main point is that all the network communications of the TVM can be monitored/intercepted.

(5) Now bring over files to be processed using the application onto the TVM, modify their dates as desired. Turn off network access and then run the application. When the application has finished, transfer data out of the TVM
and delete the temporary-virtual-machine (TVM) that ran the application.
This is accomplished simply by deleting the TVM file
This wipes out any state information the application may have kept.

(6) So every single time the applications thinks it is starting afresh. As a result attempts to control the use of the software are defeated.

Since any local storage of state information is useless, there are two other paths the application writers might take
(1) Try to distribute/hide the state information in the user data files themselves (possibly even using stegenographic techniques) (2) Communicate the state information to a trusted "Activation/Authentication" server (Henceforth denoted as "AS") using the precompiled public key of the AS.

Next We show how each of these approaches can also be defeated

I.4 Problems with hiding state information in user data

(1) Note for that any application to be of any practical use, it must be able to "import" data not necessarily exactly in the format the application writes it out.

(2) So as long as the user can manage to locate and delete the state inforation the application wants to write into the output file, the application can be told to "import" the stateless data at the beginning of each session.

(3) Stripping the application specific state information from user data files is even more easy if similar competiting software applications exist. Then for inter operability reasons, the vendors are likely to support an application independent data format. It suffices to export all data files in this application independent format.

(4) Even if such such application independent data formats do not exist (i.e., only that application vendor has a monopoly and a stranglehold over all data formats), it is still possible to figure out where the application data is being written in a file. One way to discern the location of application state information is to export the exact same data set (objects, projects, entities etc.) at different times, under different uid's, access control authorizations etc. The different versions can be compared (using any number of tools including even simple ones like the "diff" command). The portions/bytes where the data files differ must be the state information. All such differeing bytes could be easily identified and rewritten in any desired manner (since they are not encrypted).

(5) If the appliciation uses encryption or obfuscation (for instance via hashing) then it must rely on static/precompiled algorithms as well as keys. The particular aspect of static/precompiled keys which is of interest here is that the keys are always the same. Ideally the keys should be derived from hard to replicate (and in that sense close to "truly random" parameters) such as the number of network bit errors since last boot, current cpu temperature, the number of times the mouse was moved in the last two interactive logins, etc). However if the application chooses to do that then we are back to the basic "where to store the keys" problem (since storing them within application binary itself or in any other place in the virtual machine is useless because the entire virtual machine could be discarded after each use).

That in turn implies that all copies of the application that are distributed/sold must use the same key (otherwise user A cannot share files with another user B at a different site). A few keys deployed on a large scale will invariably leak out eventually.

(6) Encrypting user data leads to other serious issues for instance it is easy to do a DOS attack : if a virus could simply change a bit in the user data file the resulting file would fail application's checksums/decryption tests and the user would lose all their data.

(7) Likewise, if user A in organization Org-A sends data generated by the application (that wants to protect itself by burying "state" information in user data files) to user B in organization Org-B then the

copies of application installed Org-B must be able to recognize and validate the legitimacy of the data they received from Org-A.

It is clear that embedding of application's state information into the user data will require certificate/state verification hierarchies and will therefore inherit all the well-known probelms of certification hierarchies. Creation and maintenance of such hierarchies is certainly beyond the goals most software vendors (ought to and typically do) have: "writing a good end-user application".

For all these reasons we believe that hiding the state information in user data is unlikely to be the method of choice for enforcing DRMs.

That leaves us with the only other possibility, viz., gather state information and **sent it to a trusted third party for verification/authentication**

I.5 State validation/authentication from Authentication server(s)

Based on how actual software licencing has evolved we distinguish two cases :
case (a) The state/code/key/license verification is done only once at the time of first invocation of the application.
case (b) The state verification is done more frequently, typically upon each invocation of application.
For each of these cases we show how the Reincarnation attack can easily circumvent the license restrictions.

Case a : Local state + one time communication with an Activation Server (AS)

Now consider the case where the application talks once to an external entity that verifies the license/state on behalf of the vendor. The main point is that in this case the communication with the AS is very infrequent (only once to unlock the software) and hence the vendor might be able to provision the Authentication server in-house or at least in one location where it completely controls the Server and its environment. In this scrnario if an application presents a valid "state" the AS sends the "you are blessed to run" response. The receipt of this response unlocks the software.

This license scheme is easy to circumvent:

(1) Get one legitimate copy of the software, install it in a virtual machine and simply record the transactions with the Authentication server.

(2) Every time the application is to be re-used, re-start the virtual machine **in exactly the same state**
(2-i) The application comes up with the EXACT same state (hence the exactly identical challenge/response information) (2-ii) Simply replay the recorded "blessings" from the AS and the software unlocks itself.

note that this is accomplished without touching the encryption.

Case (b) : Local state + frequent communication (at least once per invocation) with an on-site License Server/Manager (LS), which in turn communicates with software vendor's trusted Activation Server(AS)

Now consider the most elaborate case where the application talks to a License Server/Manager every single time it is invoked and also intermittently at random times during the execution of programs after it has gathered and analyzed random portions of USER DATA/Inputs/Interactions for that particular invocation. Note that different users must necessarily apply different inputs in order to accomplish different tasks. Consequently, the state information that the application uses is different each time, it is truly session/user-input/user-data dependent.

Sophisticated commercial high-valued software packages (such as VLSI CAD tools from Cadence, Synopsis, etc., Auto-CAD, and potentially Oracle and other database softwares) typically do this type of checking. For example every time Cadence VLSI CAD package is invoked, it contacts a license server. It also "renews" the license-lease periodically.

In order to beat this scheme, realize that the license server (LS) that so closely tracks the usage of applications must be installed at the site where the application software is deployed used.

Note that the the Authentication Server in the previous case could be located at vendor's site, i.e., away from the client sites. However for the reasons stated below a license server has to be installed on each client's site.

a : Data sensitivity/secrecy
organizations like the the defense/security agencies, banks (and in fact all autonomous systems) typically go to extraordinary lengths to minimize data flows across their borders for for obvious reasons.

b : Scalability
if each time a user invokes the application (where each invocation must be blessed by a license server), if an encrypted challenge were to be sent to one central license manager at the vendor's site, it would become a single/central point of failure. Such a server could be subjected to DDOS attacks to prevent ALL clients from using the application.

c : Initial startup delay would be substantial

For all the above reasons, it is fairly accurate to assume that the License Server (LS) will be installed at the client's site. This is the weak-point that can be exploited as follows :

(1) Install License Server (LS) itself on a virtual machine. call this the PLSVM (Pristine-License-Server Virtual Machine).
(2) The application software is also installed in another Virtual Machine, the Pristine-Virtual-Machine (PVM).
(3) Note that PLSVM and PVM are simply FILES $\Rightarrow$ they can be copied and/or packaged as required.

(4) The very first time and only one single time the License Server (LS) is instantiated legitimately in "THE REAL" reality (as opposed to a virtual reality). In other words the License server is allowed to communicate freely with the software vendor's Authentication Server. However, all its network communications are recorded. (what matters is that the "blessings" from the software vendor's authentication server (AS) that "unlock" the license server are recorded). Note that there is no need to break encryption or reverse engineer or even touch code and/or data.

(5) To circumvent license restrictions, always distribute two machines i.e., the 2 FILES
(i) PLSVM and (ii) the Pristine Virtual Machine (PVM), that encapsulates/traps the application software itself

(6) To run the application, first make a temporary copy of the pristine license server, call it "Temporary License Server Virtual Machine" (TLSVM) and also make a copy of the client-machine to run the application itself which is the Temporary Virtual Machine (TVM).
Each time the software is to be used, first unlock the license server (inside TLSVM by replaying the recorded "blessings" from the Authentication Server (AS)).
Note that the replay attack on the License Server machine works because of the same reasons as above: It is started with the exact same initial state every single time. Consequently it sends the EXACT same encrypted challenge back to the Authentication Server EACH TIME. The Authentication's Server's responses are recorded and replayed from then on forever in each **"Re-incarnation"** of the License Server.
(7) After the license server wakes up, start the TVM that contains a copy of the application.

Note that both of these copies are essentially trapped in a "virtual capsule". Now the client merrily talks to the server using the *heavy encryption*. Little does the client know that the license server is nothing but simply another virtual machine running inside the same box !
(8) Once the application is finished, transfer the data out and delete copies of both the Temporary License Server Virtual Machine (TLSVM) and the client (Temporary Virtual Machine) that were run.
(9) **Any state the License Server and/or the client maintained is thus wiped out.**
(10) Since before each use we start from a pristine copy of the license server as well as the client, this new virtual-environment attack can fool the license server and its clients forever.
(11) In case the local license server wants to talk to the AS once in a while, those responses (from the AS whoever) can be recorded and played back. once again because the license server is being started afresh, replay attacks work.

In summary we have demonstrated a new method of attack : **The Re-incarnation attack $\Leftarrow$ a specific Blend of Controlled instantiation of software entities, selective "Replaying" of some parameters/messages and "Man-in-the-Middle" attacks**.

It is more powerful and more damaging than previously known attacks as illustrated by the following scenarios.

1 → **Re-incarnation attacks and entertainment industry**

Consider the on-demand movie watching scenario: The movie is dynamically encrypted with a new key for each request to play it. Furthermore, the key is typically changed several times within the duration of the video delivery.

No matter how many times the keys are changed, the Re-incarnation attack can capture the video with low effort as follows :

$\mathcal{A}$ Install the video player application (say the DVD player) in a virtual machine (call this machine the Pristine Virtual Machine, PVM).

$\mathcal{B}$ Only for one time, watch the movie legitimately.

During the legitimate run, record all the communications between the video-player-application and all the external peers it interacts with as seen by the network card (including the video stream as well as the "blessings"). Then bundle them together. This bundle of recorded communications from the server-side entities is henceforth called the "Captured-Server (CS)"

$\mathcal{C}$ "Captured Server" + Pristine Virtual Machine are encapsulated in another-virtual machine (which we call the Dragon-Virtual-Machine that simply devour any content that it comes across only once.

$\mathcal{D}$ To watch the movie for free for ever thereafter, simply instantiate the Dragon Virtual Machine (DVM)

All the DVM does is that it "awakens" the PVM in the exact same state each time and then feeds it the video stream and the "blessings" needed to unlock it.

$\mathcal{E}$ Note that the DVM is simply a FILE. In fact the DVM can be realized as a self-extracting archive so that all a user needs to do is double click to watch the movie.

$\mathcal{F}$ The ultimate blow: A virtual machine can automate the entire multimedia stream capture process ! i.e., perform all the recordings as mentioned above and pack the CS and PVM and generate the self extracting archive DVM aka, the free copy of DRM restricted content

This we call the Evil-Virtual Machine (EVM). EVM is the "script" that any "script-kitty" can execute : All the user does is starts the EVM and watches the movie only once. The EVM generates a nicely packaged DVM = free copy of the protected software and data.

2 → **Creating Free Copies of Licenced Softwares**

Now consider software applications and their use in foreign countries (China/India/..... One legit copy of software (such as Auto/Cad or Vista) would be purchased and run legitly for a few days and then "the software can be used for free by all others".

Again one Evil Virtual Machine could be dedicated to devour one target application.

3 → **Resurrecting Old License Manager Processes/Machines Again** We actually accomplished this task for a commercial VLSI CAD software suite. (purely for the purpose of Research and Development, after the experiment we destroyed copies of the apparatus).

1. Cloned the old license server machine as VM1 (Virtual Machine 1) and installed the old/expired license server from 2006 and copied the license files.

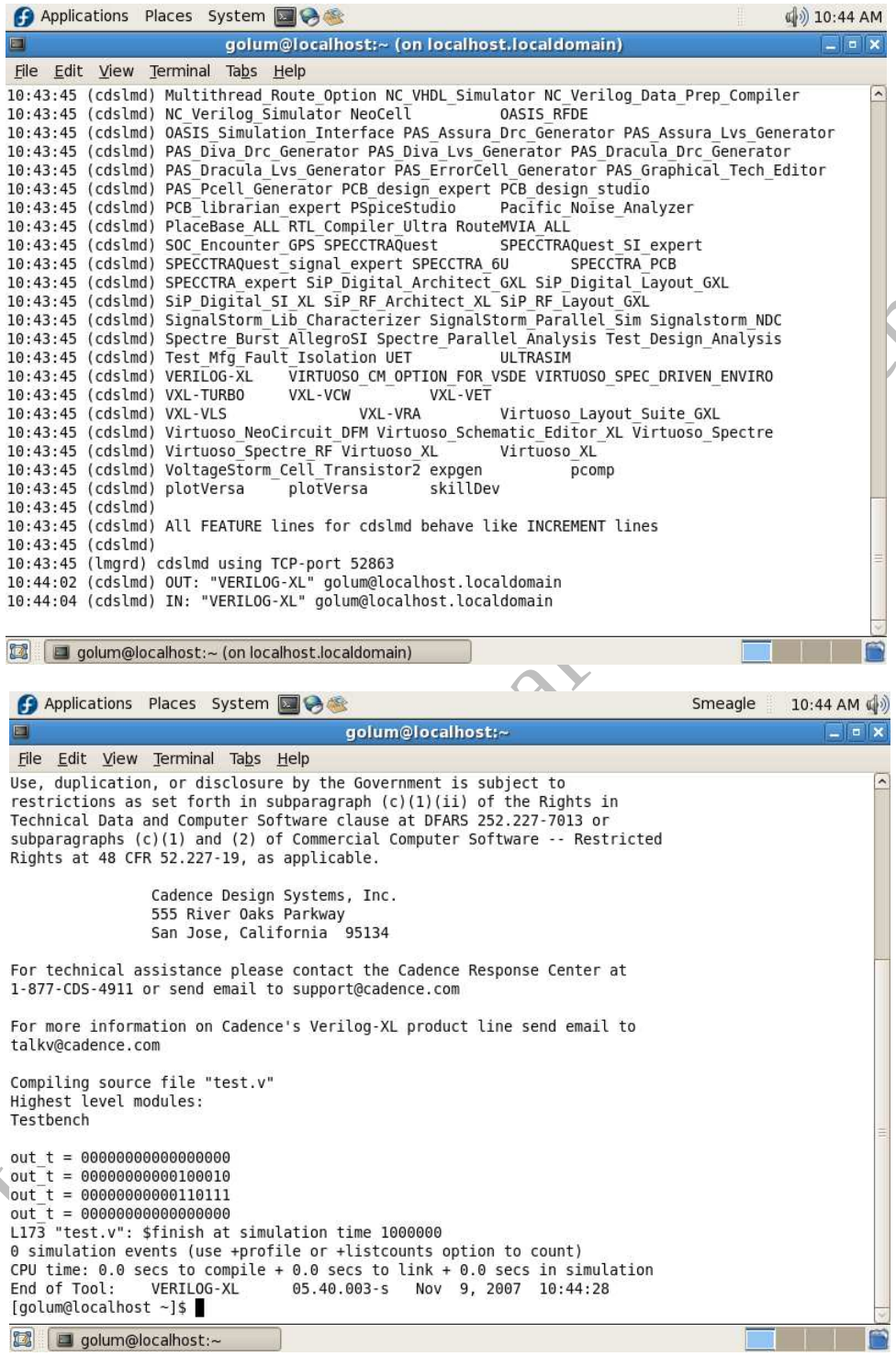

Figure 1: Screen Dumps from the License Server resurrection experiment.
Top: Re-incarnated License Server Bottom : Virtual Machine 2 running application

2. Installed the executable on VM2.

3. Set up private virtual network between VM1 and VM2. In this "virtual reality" anything can be controlled so the clocks in the virtual network all said it is 2006 years past A.D.

4. Executed verilog test case on VM2,

5. Checkout of license shows up VM1.

The Verilog code ran fully and completely on VM2 with the "blessings" of the resurrected license server. Screen dumps showing the runs are included in Figure 1.

4 → **Impact on Military Scenarios** is discussed below toward the end of the next section.

Implications

① The above examples demonstrates that the virtualization technology when leveraged toward **this new mode of attack, is a lot more capable than what "merely setting clocks back" can achieve**.

It neutralizes all attempts to authenticate-and-permit the use of software based on local state information (i.e, information local to the client's site)

② Note that "Nonces" is fundamentally a way of distributing state information. Instead of only one entity (the server side/entity) taking on all of the burden of authentication, part of the load can be balanced by distributing some state information to the client-side (typically in the form of "nonces" (ex., cookies))

③ However, the first above shows that with the help of virtual machines; such nonces or in general any kind of state information can be wiped out without breaking the encryption. Consequently, **at each instantiation, the application must make decision only based on what it can access in the current instantiation,** it cannot use state information from past invocations.

④ This in turn implies that "MMW" detection mechanisms must be built into each application that is serious about protecting itself.

⑤ Furthermore, the MMW could get evolve into a leaner and meaner entity (this is always the case in the spy vs. spy game, as long as one is ahead of the adversary, they have the advantage). As was demonstrated with the "bluepill" attack, the MMW could "turn the malicious mode on" not necessarily from the start but at any time during the current instantiation of the application. This in turn implies that **tests to detect the presence of MMW must be spread out throughout the time the application process is alive.**

⑥ In more general terms, **any remote system that can be initialized into a predictable state is potentially in danger of being commandeered away**. Virtualization happens to be one technology that can ensure that the remote system (installed at the client's site) can always be initialized into exactly same state each time thereby making it possible to use it indefinitely without breaking encryption.

⑦ **Software by itself can never uniquely identify/tag itself. This has two aspects :**
(1) Since software can be copied/replicated one replica/copy cannot be told apart/distinguished from another when in storage (i.e. when the s/w is not running).
(2) By inputting identical parameters the s/w can always be guaranteed to behave the same way ⇒ s/w by itself cannot identify or distinguish another replica/copy of itself even when it is instantiated (i.e., running)
Consequently, software is is always vulnerable to the Re-incarnation attack

⑧ To avoid Re-incarnation attacks, the chain of trust must be bootstrapped in hardware. Note that hard-

ware attributes (noise in ckts, etc) are hard to replicate ⇒ Hardware-rooted random number generators can provide a sufficiently unique/non replicable sequence of nonces.

Let us re-examine some military scenarios in light of the above

(1) Future sensor networks will be created by "air dropping sensors". The sensors then discover each other and form a network. For the ease of discovering their peers, it is desirable that they initialize into a one of the (relatively few) known initial states.

However, such a design carries the security risk we've demonstrated. If the adversary can initialize the sensors into a known state then the sensors could be "activated" and harnessed by the adversary.

(2) Unmanned reconisance agents (includes small drone planes and other types of unmanned probes) carry even bigger risk. If they can be initialized into a known state, then the adversary could commandeer them to the adversary's advantage.

******** parts after this are being re-written. the second part deals with how to solve the malicious middleware problem

——- IGNORE STUFF AFTER THIS LINE for the time being ————-

II The fundamental problem of MMW detection and our solution strategies

As seen in the prior sections if a license server can be installed on a virtual machine, the software that the license server intends to protect can be freely used, copied and distributed to others which corroborated the well known quote/adage "a bit in control of the user is not a bit under your control" [10].

In essence Malicious Middle-Ware (MMW) can subvert the application's sense of reality by trapping it in "virtual reality" environments. and inducing it to unlock itself.

∗∗∗ The problem is then essentially this : how can an application decipher if it is being subjected to the MMW attack ?

First, we would like to point out that MMW detection in its broadest sense is distinct from Detecting the presence of a Virtual Machine server. For instance the MMW would include the traditional Man-in-the-Middle attack (which can happen in the network, far away from the virtual machine that runs the client).

In security everything depends on the assumptions made ("security is a state of mind..."). We therefore clearly state our assumptions first.

II.1 Assumptions

(1) The Malicious Middle Ware can completely isolate and encapsulate the application and the License Server entities.

(2) MMW can set any parameters it wants in this virtual reality trap in which the license server and the client software are imprisoned. Consequently if the application queries "current time" the Virtual Reality will give it the time the application would like to hear.

Note that it is not merely a matter of software trapping alone, the virtual reality could include physical hardware machines dedicated to run the "canned" license server and application software. The actual hardware clocks of these machines are initialized to whatever values the owner of the virtual reality wants them to be.

(3) No attempt is made to break encryption or reverse-engineer the code. In other words we are establishing the capabilities of Re-incarnation (i.e., a blend of controlled re-instantiation, recording/reply and man-in-the middle) attack by itself.

II.2 Possible Solution Strategies

This second assumption above implies that the application software cannot trust any response it gets from BELOW (in the direction of the CPU and network interface(s) of the machine on which the software is running.

Neither can it depend completely on inputs it gets from the USER (in case the user is actively interacting with the software (which may not always be the case, for instance Oracle servers), the MMW could very well playback/filter/modify/delay user-input events (key-clicks, mouse movements, etc) just enough to thwart any quantitative estimation of real-time intervals and other parameters which could help break the virtual reality trap.

(1) Pre-Compiled Core of trust (Uncover Virtualization Wizard (UVW)

One Possibility is to include a Pre-Compiled Core of trust. We call such a core the "Uncover Virtualization Wizard (UVW)". It consists of a suite of a large number of executables of all different typed (predominantly memory bound, CPU bound, CPU blaster, memory blaster/hogger, network-card blaster, i/o bound, mixture of all, etc.). The application software would randomly chose a workload mix and run it. and keep track of the relative order of completion of the tasks as well as the estimates of the amount of time required to complete each thread in the workload.

In order to hide its own existence, the MMW (which creates the virtual reality capsule) must block/heavily filter at least two things

(i) System calls that return real Time Durations as well as those that current date/time values.

(ii) If the application can get an encrypted message across to some trusted third party "time stamper" the virtual reality siege can be broken. To prevent this MMW must filter all communications with external entities.

second item above is where the MMW can be overloaded. For the purpose of illustration, assume that there is an infrastructure of "relay nodes". All they do is decrypt the incoming packet with their private key and uncover the real IP address inside and forward the "inner payload packet" to that address. Now the application can be pre-compiled with the public keys of at least a few well known relays. It blasts a flurry of packets at random times to a bunch of relays. The inner addresses of the payloads all point to the software vendor's Authentication Server. The relays decrypt the payload and send the packets to the AS. The AS then sends the replies back through a different set of relays. This way the Malicious Middle Ware will have hardly any clue about which of the outgoing packets to block and which of the incoming packets to block, if it tries to filter all, it gets overloaded and the application can lock itself down.

(2) "Hardware Assisted Lifting of Trust"

Note that the noise in a circuit and other physical attributes are very hard to replicate. As a result a hardware-rooted random number generator that uses underlying physical quantities (thermal, electrical noise, residual battery charge, etc) to generate the random numbers produces a sequence that is hard to replicate. Samples from such a sequence could be used as **seeds** to a regular LFSR or Fibonacci feedback shift registers. Since the initial seed is unique the resulting sequence of random numbers will also be unique, almost impossible to replicate.

Furthermore, hardware cannot be "copied". Because of these two reasons, hardware can uniquely tag/identify/name itself.

We therefore propose the following method hardware assisted method to defeat Reincarnation attacks (we would like to re iterate that one of the assumptions is that we do not want to touch or reverse engineer code or break encryption. We are finding out how far Re-incarnation i.e., a combination of controlled re-instantiation, recording/replay and man-in-the middle) can go by itself.)

(1) The license manager is a physical hardware module (such as a smart USB dongle or a smart card). It stores a number of public,private key pairs and it can internally generate private keys.
(2) For each copy of the software suite to be licensed, a physical key (in the form of a dongle/smart card) is created. and physically mailed/sent to the end user (this is the one-time out-of-band communication taking place via the good old "sneaker net"). The physical dongle contains a sufficient sized EEPROM and encryption/decryption modules as well as the two most critical components (that together are the unique features of our proposed mechanisms to the best of our knowledge)
    - A hardware rooted random number generator whose output is very hard to replicate.

- Entries in the PROM that store the date and time when the dongle was created, to what day it is supposed to be used and maximum number of invocations (and possibly a few other parameters).

(3) After receiving the physical key the user inserts it in the USB-port. The key/smart dongle then gives the user some codes that are used to authenticate the user to the License Manager Vendor (LMV, which is the entity that plays the role of Flex LM or other such vendors). The LMV then embeds/encrypts the application with public keys whose corresponding private keys are known only to the dongle (not even to the LMV employees/entities that are authenticating the end user).

(4) Now when the application is invoked the first time, it sends the dongle an encrypted value of the current time reported by its environment.

(5) The dongle reads it, decrypts the time inside, on chip, compares time-stamps, if valid stores the new time-stamp physically in the EEPROM that only it can see, because it is physically contained within the smart dongle. It may overwrite previous time stamps as required)

(6) The dongle then re-encrypts the time and a random nonce created by the hardware-rooted random number generator with one of the public keys of the LMV's Authentication Server, and returns it to the application.

(7) The application has to pick up this payload and send it to the Authentication Server. The Malicious Malware is now powerless because all the encryption/decryption is happening inside the smart dongle. Also replay attack does not work because the nonce is hardware rooted and therefore not replicable.

(8) The AS of the LMV compares the time-stamps and performs a predetermined function on the nonce and encrypts it with another public key whose private key is stored only inside the dongle. When the application receives this packet, it re records its own time stamp again and sends the both these to the smart dongle. The dongle then sends an encrypted "yes or no" reply to lock or unlock the application.

We once again point out that the assumption is that the code is not being tampered with. The current secure execution platforms such as Intel Trusted Execution Technology [4, 5] and AMD Secure Execution Machine provide assurances that the code is correctly loaded and has not been modified. We believe that the only thing missing was the ability to bind the instantiation of the executable to the hardware key. This functionality is achieved by the proposed mechanism under the previously stated assumptions.

In the most general sense, the only way to break out of the "virtual siege" mounted by the MMW is to have an "out of band communication with some trusted entity".

The proposed hardware key mechanism achieves such an out of band communication effectively via the "sneaker net." We believe it should not be difficult to substitute this physical path by an electronic delivery mechanism. We are currently working on developing this technology.

III The "license Stretcher" attack

Some applications go to extra-ordinary lengths to include hardware keys in the forms of USB dongles or other hardware modules. As mentioned above, **if designed the right way**, the smart dongles do thwart Re-incarnation attacks, i.e., software will not function without the hardware key and multiple instances of the same software cannot concurrently run either.

In many cases, software revenues depend upon the number of concurrent (and/or independent) copies one is allowed to run. (typically called "the number of seats").

The License-Stretcher attack is in principle very simple: Instead of buying *n*-seat licences, buy a single seat license and still allow multiple concurrent usage.

The method is simple :

1. The application is always instantiated on a machine where the hardware key is deployed and is always instantiated in the name of a default owner (in case the dongle uses UID values to check for multiple concurrent instances).

2. When User 1 wants to execute the application their files are brought over to the application server machine; a new window/tab is created and bound to the first virtual desktop which is mapped/piped back to User1's actual desktop (if User 1 is Connected via virtual network computer)

3 When User 2 wants concurrent access, their files can be brought over to the application server under the same default ownership but in a different directory/folder.
The wrapper then requests a new window/tab, binds it to the second virtual desktop which is mapped back to User 2's physical desktop.

4 Multiple users are handled the same way.

There are some performance and security issues : what if user 1 and user 2 use clipboard at the same time ? We are in the process of demonstrating that it is fairly easy to extend native OS's support for multiple concurrent clipboards.

Performance is likely to be less and less of an issue because the hardware manufacturers are themselves putting in the hooks for efficient virtualization. In other words the job of the hypervisor is becoming easier as it becomes an ever thinner entity that simply doles out/maps tasks to hardware components. (for example two physical duel core machines might support say 6 concurrent virtual machines). With increasing hardware support for virtualization, the slowdown caused by concurrent usage by multiple users should be expected to dwindle rapidly, making the License-Stretching attack ever more attractive to small entities with close-knit work groups where the individuals need to collaborate anyway. In such a case instead of buying multiple licences the small company might be able to get away with a single seat and use the money saved into more hardware resources for the single seat server machine instead.

Virtual Machines make it easier to reverse-engineer software

Note that Virtual machines provide a convenient simple and efficient mechanism to snapshot/single-step/debug the entire state of the processor. If everything was true hardware, it might be harder to record/snapshot/single step the entire system.

In as sense, virtual machines constitute a more versatile/powerful and easy to use "de-bugger". Un-rolling, reverse-stepping is also possible in software (note that hardware debugger can only remember past values in variables, it may not be able to actually "undo" say the last k operations. A virtual machine that is fully emulated, on the other hand is simply a file, and hence, a REVERSE MACHINE could also be constructed in SOFTWARE to back-track to desired points. In other words Forward and Reverse operations to rapidly converge to the desired state are analogous to double sided convergence (where the error between desired and target value can be both positive and negative) whereas a hardware processor can really only execute operations in one direction, which is like single-sided approximation.

Consider another example: If the operating system always loads itself in the same way and locations, then the adversary knows what locations need to be compromised. If on the other hand the OS dynamically loads itself in different locations at each boot time then the job of the adversary is harder. One possible non-replicable method would be deliberately load random values in the memory after the memory-check at boot-time is finished. A malicious hypervisor underneath the OS could change the contents of the memory to be all the same and then the OS "thinks" it is dynamically loading itself but the hypervisor is forcing it into a any box it wants....

IV Conclusions

In this paper we presented the novel Re-incarnation attack. We showed that any license and DRM method that depends solely on software can always be circumvented. Actual screen dumps of experiments done to verify the theory were presented. Besides the fundamental theoretical observations/connections made throughout the paper, we conclude with a speculation about philosophical implications of virtual machines:

One of the most famous bad theories is the Cartesian belief that the mind is software, separate from the body which is hardware. This is definitely not how humans work, and this bad theory leads to horrendous philosophical problems.

Now along come virtual machines (VMs). Typically these are defined as existing entirely in software, completely separate from any hardware. As such, VM is a Cartesian "mind." What are the implications?

Our work demonstrates that software cannot uniquely identify/tag/name itself.

However an individual's mind is quintessentially their unique characteristic or in other words mind can uniquely identify/tag/name itself, no two minds are known to be exactly alike.

This suggests that Mind cannot be a purely software entity. (since mind can uniquely identify itself but software cannot do the same). Mind games and debates aside, we believe that virtual machines can never achieve "self consciousness".

In other words we believe that a fundamental contribution of this virtual machine work is to better delineate what is required to achieve "Self Consciousness"

References

[1] e. Samuel T. King, Peter M. Chen, "Subvirt: Implementing malware with virtual machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, (Oakland, CA), 2006.

[2] S. Crosby and B. D., "The virtualization reality," *ACM Queue*, vol. 4, no. 10, pp. 34–41, 2006.

[3] "Tcg specification architecture overview," tech. rep., Trusted Computing Group, August 2007.

[4] S. Campbell and M. Jeronimo, *Applied Virtualization Technology*. Hillsboro, OR: Intel Press, 2006.

[5] D. Grawrock, *The Intel Safer Computing Initiative*. Hillsboro, OR: Intel Press, 2006.

[6] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ: Prentice Hall, 2003.

[7] P. Ferrie, "Attacks on virtual machines emulators," in *9th Annual AVAR International Conference*, (Auckland, New Zealand), December 2006.

[8] P. A. Karger, "Multi-level security requirements for hypervisors," 2005.

[9] "A proposed interpretation of the tcsec for virtual machine monitor architectures," tech. rep., Trusted Information Systems, Inc., 31 March 1989 1989.

[10] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley, 1994.