# Immutable Runtime

This is just an excerpt from an idea I had for computer systems with great longevity that appears more and more useful in its own right as the days go by. I thought of it as a way of making normal workstations - primarily those used for clerical work - extremely resistant to corruption by way of virus or Trojans. Now I get the feel that this might be more useful for high integrity systems or systems exposed directly to the internet.

First some background: a computer actually performs its work in the processor. It recognizes two forms of data:

1. Instructions.
2. Operands(usually just called data).

It reads these two forms of data from main memory, on a normal computer consisting of one or two high speed memory banks. It's not easy to get an operating system to take information from some remote server and run it on the local machine. Operating systems are written to be very careful about that sort of this. The trick used by countless viruses and worms is to get placed in main memory as data. Operating systems are much more tolerant of data from remote sites being written to memory.

This works because computers don't really know the difference between instructions and data. It only knows that blocks of 32 or 64 bits of information can be retrieved from main memory and placed in various containers. Unless I as a programmer make sure my program will only write operands to areas dedicated for storing them, there is little to stop some attacker from overrunning the data-area and modify the software being executed.

Unfortunately it is a bit more complicated... A lot of attacks succeed because memory can't be allocated when the programmer writes his code. The software will itself have to create interleaved blocks of pure data and other blocks that contain instructions. So how do we get around this? One way is to introduce a No-eXecute(NX) flag in the computer's memory that prevents pure data from being interpreted as instructions. I'm moderately impressed, not least since the NX flag is set by software, and if we could trust software to be correct we wouldn't see much need for this feature to begin with.

Another solution is to use a programming language like Java where I as a programmer am not even allowed to fiddle with memory allocation like that described above. There is a virtual machine that handles that for me, whether I like it or not. This works well because dozens of hardened programmers at Sun Microsystems(now Oracle) can write that kind of software correctly whereas I can not. So every Java program enjoys the benefits of the expertly written memory-handling service in the Java virtual machine. This kind of feature used to be seen as slow but today it can sometimes be faster than the way of doing things where every programmer had to handle his own memory allocation.

Now - at last - we come to my idea of immutable runtime. It's actually quite simple in theory. Let us declare that the main memory of a computer can be divided into two contiguous sections that do not overlap or leave any gaps. The first section is called X and the second section is called D. We enforce three rules for these two sections:

1. The processor can only fetch instructions from memory addresses located in section X.
2. Before any instruction can be fetched from section X, it must be set as read-only.
3. Once section X has been set read-only, section X can not be expanded to cover memory addresses previously in section D.

While it isn't specified above, section D can be modified as usual. The specification does allow for memory in section X to be relabeled as section D but I would advise against it. Firstly that allows for modification of the partitioning of memory. While we only want to allow the safe modification whereby we reduce the size of X, we should be careful about relying on our ability to determine what "safe modification" means. Secondly we may well have allocated section X certain space for a good reason and have no business reducing it.

Some questions you might have: How is main memory populated with instructions before locking down section X? Obviously we can't rely on the operating system or even the boot manager to perform this work. Both of those rely on the processor fetching instructions from main memory. I suggest that the process of populating section X is handled by custom circuitry. It need not be difficult. I suggest an SSD device into which we put our executable instructions, setting a "valid" flag for those addresses containing proper instructions. Then we can upon boot copy instructions from the SSD device into main memory. The last address into which it writes executable data is chosen as the last address in section X.

To enforce strict control over the executable data the SSD should have a Write-Enable switch that is physically operated. That way an attacker can't place his virus in that device unless he has an accomplice standing next to the machine with his finger on the WE button. Once section X is set as read-only the regular boot procedure can begin. I have just had a change of heart which I can happily say doesn't require any modification of the three rules laid out above. I have always thought that section X should only contain instructions and no operands, but it is sensible to have some operands in section X. If it were otherwise we would have to populate section D using the same - or at least a similar - system we use for populating section X. The kernel, boot manager and all of their constant operands are loaded into section X and this gives <u>them</u> the ability to populate section D with data.

One could go as far as saying that all constant operands for all software should be placed in section X. This is not a bad idea, the only immediately apparent drawback being that some constants might not be needed throughout all the execution of a program and so it would be efficient to delete it once it has been used. Of course efficiency is not the dominant theme of this system. It lacks efficiency and flexibility. Today people don't know what software they're going to run on their computer when they boot it up. They can be confident that their operating system, e-mail client and office suite will be used, but what about web-browser plug-ins or media players? We solve this by taking chunks(pages) swapping them between main memory and hard drive. Something quite different from what I'm proposing.

The idea for immutable runtime stems from my LCSC-project, which relies on a very ascetic suite of software which can fit into a computer's main memory with room to spare. As described above, it is not feasible to use this system for my current workstation. The NetBeans IDE takes up several hundred megabytes of RAM. Steam itself isn't a big drain on memory but the games I launch from Steam need a lot of room. With the immutable runtime I would have to modify the SSD containing software I want to run and then reboot the computer.

The drawbacks of my system and the impressive work by professor Nickolai Zeldovich at MIT made me think that it had no *raison d'être*. Upon closer inspection, Zeldovich's systems are complicated to use, not unlike SELinux. He does offer stronger guarantees than SELinux and possibly fewer areas of confusion, but it is not a trivial system. If you want to deploy a server with an immutable runtime I think it can be quite straight-forward.

1. Boot the factory-default system(a modified Linux distribution perhaps).
2. Select software to include in the live system.
3. Begin install.
4. Press button to modify SSD containing the software to run.
5. Upon completion of install, reboot.

For ultra-reliable thin clients or LCSC-style clerical machines used by people who just want a kind of electronic typewriter, the benefits of this system are even greater. Unless you want to modify the software you won't even know that the device has an immutable runtime. Perhaps the person who sold it to you mentioned it and explained that it makes it impossible for an attacker to compromise your system without your explicit approval. That last bit is not a marketing ploy but a real caveat. If an attacker sends you malware and asks you to install it, and you then go through the procedure for installing new software, then: yes, your computer will be infected with malware. It is not an immutable system. The runtime is impossible to modify when it is running but you can change what goes into the runtime environment when you next turn on your computer. This can be mitigated by only allowing software to be installed if it has been cryptographically signed by the people who manufactured the computer, but that is a separate issue.

How is this to be implemented? I've considered various methods but as Jamie on Mythbusters says: simpler is always better. To enforce the first rule, prohibiting data outside section X to be fed as instructions to the processor, we place a device that masks addresses given to the processor's program counter(PC). Let us say that the highest address from which one can fetch instructions is 00001111. If we apply the AND operation on any incoming address with the mask 00001111, the four left-most bits will always be zero. So address 00000100 will end up being unmodified: 00000100. If we choose a number just two steps above the last valid address, we get a different story: 00010001 becomes 00000001. This has a minor problem, namely that we can only split memory into sections X and D on a multiple of 2. So our mask above states that section X consists of the first 16 addresses. In reality of course we won't use 8 bit addresses so we are more likely to make section X 128 MB in size. If the software requires 129 MB of main memory, we have no choice but to change our mask and make section X 256 MB in size.

If this is too rigid it's not a big issue to use a simple comparator and possibly an arithmetic unit. Then we can say that section X consists of the first 129 MB of main memory. Any address fed to the program counter is first compared with this fixed number. If the address is above 129 MB, we can either throw an exception, halt execution or fail gracefully. To fail gracefully can be to mask out the bits that definitely put us over the limit(works well if the offending address is the result of a flipped-bit error) or shift the bits in the address to the right until it's back in the legal range(in case the programmer forgot some arithmetic step). Of course we can be even more flexible and declare several separate sections of memory as making up section X, though I don't see much point in that. The important thing is that the mask or the memory device containing the highest allowable address is made read-only before the first instruction can be read.

How then do we keep people from writing to section X? Obviously we should use the same source as before, whether it be the mask or the stored value(I differentiate them because the mask need not be readable while the stored value would be). The instruction for writing data to memory should have to pass through the aforementioned check. If the target address is above section X, the write goes through. If the target is within section X we handle it as we see fit. This is easy to do with a true RISC processor like MIPS but it can also be done for the x86 I think. Even the x86 has some hardware which pipes data from registers into main memory. Wherever that is, we splice the control mechanism in there. If we don't want to put this into the actual processor we can place it closer to the chipset handling accesses to memory. That makes it trickier to handle violations cleanly(we could issue an interrupt which isn't half bad) and also means we have problems with caches. We don't want the processor to be able to circumvent our barriers by using on-chip memory. Thus it seems necessary or at least prudent to integrate these safeguards into the processor.

Is there any software written with the understanding that instructions and data are separated as described? Not to me knowledge but it doesn't seem like a major issue. A compiler can be created that makes sure all operands are placed in section D and instructions in section X. Even hand-coded assembly can be analyzed statically to see how it can be split into two sections. One problem is that MIPS amongst others doesn't allow for arbitrarily large jumps from instructions and operands. This means we either have to choose an architecture which avoids such restrictions or modify one which has them. Today a lot of processors are built around IP blocks, where IP means Intellectual Property, not Internet Protocol. You can buy the schematics for an ARM or MIPS, modify it to suit your needs and then get someone to manufacture it. In a pinch we can use binary translation à la Transmeta and replace instructions that require jumps that exceed the architecture's capabilities. That would however go against the simplicity and robustness the system tries to guarantee. I guess my statement about not understanding why one would want to interleave X- and D-sections no longer holds true. If you want to break up a program into distinct parts but still use a standard instruction set that doesn't allow loads or stores to addresses more than 512 MB from the current address, it would make sense to create a set of sections X and D in succession. It could be a 12 MB section X(instructions) and a 17 MB section D(data). Then another X and another D. All enforced using addresses placed in write-protected memory upon system boot.

Let us consider another variation of the immutable runtime. Imagine that some people running high integrity servers like this idea of immutable runtime. It offers strong guarantees and isn't particularly complicated. They do however want to be able to expand the size of section X during runtime. They need to include new software or simply add more instances of the applications currently running. Well, we can't satisfy the original rules of an immutable runtime, but that is not a goal in itself. So how can be make the rules less rigid without introducing too much vulnerability? Number one is to maintain the electronic lock on whatever device defines the border between sections X and D. We do however make it possible to override it manually, just like we do with the SSD containing the software we want included in the runtime. So if you want to add another ten instances of some web-application you first tell the operating system to stop using the parts of section D that border on section X. They are of course to be populated with instructions for these new instances. Once the memory is free the operating system moves data into it. I don't consider it necessary to go by way of the SSD, we're already guaranteeing a human operator is right there pushing a specific button. Once the segments are populated we hold down the button that allows us to change the limits of section X and tell the operating system to enter the new value. We release the button and we have our new runtime which enjoys the same guarantees as usual.

Unsurprisingly the big issue here is that we need to make sure the new software that we bless by including it in section X is in fact safe. We can use the idea of cryptographic signing or a static analysis of the software we are about to include. Does it make attempts to call parts of the kernel that it shouldn't? Does it try to modify data that it didn't create? This static analysis isn't fool-proof but it can go some way to help operators avoid dangerous code. Keep in mind that the immutable runtime by itself doesn't guarantee that information in section D is kept safe from malicious parties or incompetent programmers. Nor does it offer any such guarantee for the file-system. It merely guarantees that the code separating applications isn't modified by an attacker. If it was bad to begin with and will happily erase any file pointed to be some memory address in section D, then the immutable nature of the runtime won't stop an attacker from deleting all your files. The same is true for information in section D. A further example would be an e-mail server. If an attacker gains root access to such a server he can change the memory defining which users are trusted and what kinds of things those users are allowed to request. The executable code doesn't change because no one is standing over the server holding down the right button, but the tasks it performs depends on data that the attacker can change. An e-mail server with immutable configurations and access control lists isn't sufficiently flexible so that isn't much of a solution.

For the above scenario to come about the attacker would have to figure out a way to log in as root using proper mechanisms. He could not engage in privilege escalation as he can't change the way the kernel does its job. He can not inject shell code through some faulty web application. So while it is not impossible to gain root access to the system, doing so by force is not a sensible approach. While an existing e-mail server can be made to send spam the attacker could not give it new functionality like spreading pirated software, carry out DoS attacks against third parties or such things. Just protecting the kernel and a Host-based Intrusion Detection System like Tripwire or my own favorite OSSEC will protect the system immensely. Even if a hacker gains access through an application not enjoying immutability, the HIDS would log the access and report any changed files, including for instance modifications to the access control lists of the e-mail server.

Is the immutable runtime truly necessary? I thought so but have come to doubt that. I have gained great faith in GrSecurity which uses randomization rather than immutability to maintain integrity. By placing chunks of memory in random order* an attacker can't know where his injected code will end up or how to reach it. The features of GrSecurity are extensive but it is worth mentioning that it can allow the administrator to protect and even hide an HIDS. Another question that bears consideration: Is the immutable runtime useful? Yes. While servers not relied upon for national security and safety in a nuclear power plant will do well using GrSecurity or even SELinux, some high integrity machines used in banks could benefit from this system. It is most useful where the drawbacks aren't even on the cards. Thin clients are never supposed to install software so there's no problem there. Similarly people who don't mind spending half an hour adding new software once a year but abhor anti-virus software, firewalls and the ever present spectre of malware stand to gain tremendously from this system.

* *The way blocks of code are placed in memory is random but the operating system of course keeps track of this random sequence. Otherwise it wouldn't know how to execute a program from beginning to end.*