



Calling For A Revolution

Overhauling antique IT patterns for security's sake

Jürgen Pabel, CISSP

Most of today's IT aspects base on decade old achievements, some of which are not well suited for today's requirements anymore. Many aspects of IT exhibit lapsed properties: computer system architectures lack true security features, software developers must implement security concepts on top of the execution environment because the systems lack the needed paradigms. Networking protocols only provide functionality for packing bits inside other bits, while today's requirements demand so much more in terms of security.

Before some technology propositions will be presented, it is important to understand an obvious, but often neglected facet of IT: developments embrace the environment's own paradigms. Applications running on UNIX-like systems usually embrace the UNIX API, which happens to reflect many aspects of the C programming language – therefore, common UNIX applications are implemented in C. Similar attractions can be found everywhere, for a good reason: intermediary layers only obfuscate the real paradigms that the product will truly operate on. Additional conflicts arise when new paradigms are added to the environment, multi threading represents a perfect example: Microsoft Windows has integrated multi threading in its core and no conceptual issues exist, while UNIX-like systems have gained threading support only as an add-on, stirring up many discrepancies (signal handling, thread status after fork'ing, etc).

Aside from technical deficiencies of certain environments, some paradigms are simply too complex to be applied correctly and coherently in commercial environments, where time-to-market dominates. A solution for this dilemma might be the use of technical means to ensure semantic correctness - that is, to require deeply seeded structural requirements before a certain paradigm can be employed¹.

Now, let's get down to some propositions – what this paper is supposedly all about, after all. Hardware forms the base of all computing, and this area has not evolved significantly in terms of security; TCPA is the only apparent exception, but its role is to secure the computing base – not to provide an environment that supports the running software in maintaining its state of security. One concept that needs to be researched and implemented is that of memory access restrictions: not the basic concept of kernel versus userspace memory access, but a more fine grained approach.

¹ what this means and how this could work is an entire research topic



It is this simplistic approach, that makes protecting data effectively in most application models very hard; a single implementation flaw is often enough to compromise the entire software, if not the whole environment. To protect against these attacks, new memory access paradigms are needed: marking certain segments of memory to be only accessible by specially designated code, after its integrity and contextual correctness have been verified by the hardware. Additional programming concepts – not API's – are necessary to enable developers to rely on these mechanisms. For example, new keywords could be used to mark certain portions of data to be protected in this manner – much like the already existing code access restrictions in many software languages (C++: public, protected, private; Java: public, private and the little known no-declaration, which works much like a “protected”). This mechanism would not restrict the calling of executable code, but rather protect data from being accessed through any means, other than the designated accessor code. A prime candidate for this mechanism is the memory region, that stores the private components of a cryptographic key or passwords; essentially, any data that needs to be protected even outside the protection mechanisms that the handling software provides. Attackers would therefore not be able to retrieve sensitive data, even if they have taken full control of the process. This capability would also be useful, even without the involvement of a malicious party: production environments are often only accessible by authorized operational staff, who may lack skills to resolve more involved technical issues. Product vendors often demand access to the systems in question, in order to analyze an issue – the proposed mechanism could protect sensitive data.

A similar concept could also be used for protecting devices against use in unauthorized environments, like retrieving data from a stolen storage device. Today, software has to implement these cryptographic mechanisms to provide this functionality, but concepts and implementations are very different and incompatible, depending on the used product. A standardized hardware mechanism that would be available for the operating system to rely on would not only increase compatibility, but also increase acceptance, because it could function nearly transparently to the user as a part of the system's security mechanism (which could be something like TCPA). Another aspect that would make exploiting software flaws more complicated on the lowest level (buffer overflows and the like) is the use of a different memory model: The Harvard memory model employs completely separated address spaces for code and data. Combined with the proposed memory access restriction capability, it would eliminate many of the technical security issues that arise in today's IT systems.

On the networking side of things, a new attitude needs to be adopted: to embed security on every layer, instead of relying on specific layers for protection. It's simply unacceptable, that physical connectivity is the only thing required to participate in a



modern network. While this allegation applies primarily to Ethernet based network technologies, it is nothing but unacceptable, given the dominance of this technology. Mechanisms like 802.1x are available, but not in commodity hardware. Another aspect is that of activation by default: much like WEP (or WPA) for wireless, these mechanisms are left deactivated, because it is expected that a standard configuration is not prepared for their use. This approach needs to be changed in the next generation of networking technologies, any security mechanism needs to be integrated as a mandatory component (this means: not deactivatable). Even if this layer provides sufficient security in terms of confidentiality, authenticity and integrity, it is important that all higher layers provide similar mechanisms, in order to provide security for all participants and applications on the entire network. Another significant aspect that current networking technologies lack, is that of dynamic security negotiation: whether data confidentiality on the packet layer is acceptable from a monitoring standpoint by the involved network providers, or if confidentiality should be restricted to specific data components on the application layer. Naturally, these negotiations need to be secured against tampering, but that's a matter of details. Any security negotiation that doesn't comply with the user's preferences would then prompt the user for instructions. An appropriate user interface is a very important aspect of this mechanism, to not suffer the same fate as the dialog box about SSL certificate warnings.

Reflection is a useful property found in many modern software development environments. How reflection hasn't entered the realm of networking APIs is beyond me. Constructing a communications profile by querying the attributes of all involved layers would not only help in determining the security qualities of a dynamically negotiated communications channel for comparison with the user's preferences, but also allow for software to adapt their networking behavior to the discovered environment. Implementing reflection mechanisms can be a large effort, if every layer has to implement the logic for the interface. However, it could be easy, if a single reflection implementation were to interpret every layer. A common data format could allow for an easy implementation and much more: all layers in networking construct and interpret either fixed data structures or streams of data, based on a certain grammar. The different formats have very seldomly anything in common, which means that a special interpreter and generator is required for each one – completely unnecessary complexity. If a universally valid data format could be established, each layer could be defined by a data format template: much like what DTDs and schema's do for XML. XML is a globally accepted data format, why not make XML the designated universal format? Because XML is text based and therefore carries a high protocol overhead, both in terms of parser complexity and data overhead. However, a good binary representation of XML could be parseable in a fast manner and should



impose only a minor protocol overhead². Each network layer could then be defined by a protocol template, and a single interpreter implementation could be used to inspect all layers. If even the layers above the real communication layers – the “application” protocols, which is a very unfitting term – were to be expressed in this format, it would become possible to inspect any protocol without a priori knowledge. Firewalls could be configured to filter all unknown protocols, while verifying correctness of those known.

The presented items are discussed further in the upcoming book “IT-insecurity: The computer industry's way of life”³, as part of the section “Technological propositions”.

2 As I recently learned, there actually is a W3C working group for Binary XML

3 <http://www.it-insecurity.com> – Targeted release date: 08/2005