What Hackers Learn that the Rest of Us Don't

Notes on Hacker Curriculum

Sergey Bratus
Dartmouth College

The hacker culture has accumulated a wealth of efficient practices and approaches to computer technologies—in particular, to analysis, reverse engineering, testing, and software and hardware modification—that differ considerably from those of both the IT industry and traditional academia. (Some groups in academia share and have influenced elements of the hacker culture, but these are exceptions, not the rule.) In particular, the "curriculum" a hacker experiences while learning his skills is substantially different from that of the typical computer science student. Yet, in many respects, this curriculum produces impressive results that enrich other cultures, and its influence and exchange of ideas with the more traditional cultures are growing. Thus, understanding and describing these curriculum and approaches is becoming more important day by day.

Over the past few years, we have seen the impact of the hacker culture grow significantly. Exploits that were once disclosed only on mailing lists or in underground magazines are now published in books,[1] while publishers such as Syngress and No Starch Press produce entire tracks of "hacker" books that differ in style and substance from the heretofore accepted formats. Classic hacker tools made their way into academic curricula and industrial training, such as the SANS Institute courses (http://sans.org), and hackers have established consulting companies that major vendors now seek out.

Ignoring or marginalizing the hacker culture means passing up unique opportunities and valuable knowledge—at our own risk. In this installment of Secure Systems, I examine what distinguishes the hacker experience from that of most traditionally trained programmers and show how hackers can improve the state of the art in practical computer security.

The hacker community

We can obtain some idea of the value the hacker community has contributed to the IT security industry from the commercial success (and admission prices) of BlackHat and similar conferences,[2] at which hackers present their results to the industry. In academia, some prominent researchers and institutions recognize this value as well. Consider that in 2004, the US Naval Postgraduate School sent its team to the "Capture the Flag" tournament at Defcon, the most popular hacker convention in the US, and won (http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1000503,00.html). More significantly, important features have made their way into mainstream software after the hacker community designed, implemented, and tested them. Examples include canary-based stack overflow protection (StackGuard) and executable memory page protection through x86 segmentation (OpenWall and PaX).

It is important to note that such contributions, which resulted in significant improvements of consumer and business computer environments, happened through public disclosure and publication of relevant research in open venues. "Black hats," who act for personal gain and without regard for possible damage, have opposed such publications because they significantly reduce their efficiency and ease of operation (forewarned is forearmed). Therefore, this discussion necessarily centers on "white hats" who are ethically opposed to abusing computer systems, and "gray hats," who might run afoul of existing laws but are motivated to warn the vulnerable and minimize damage. Summarily dismissing "gray hats" as "lawbreakers" would be a mistake: after all, even well-known academic researchers have been threatened with criminal prosecution for publishing their research under laws such as the Digital Millennium Copyright Act (DMCA), and ill-informed efforts to ban broadly defined "hacker tools" are ongoing and succeeding (see http://arstechnica.com/news.ars/post/20070528-germany-adopts-anti-hacker-law-critics-say-it-breeds-insecurity.html).

Hacker knowledge and methods

are no longer limited to the select few, and the hacker culture will undoubtedly continue to attract more participants, including students and developers. Thus, industry and academic leaders must acquire a better understanding of that culture, its values, and its unique strengths and weaknesses, either to benefit from hackers' contributions or defend themselves from malicious "bad apples" who reject hacker ethics.

The hacker methodology

Before examining the hacker modus operandi, let's summarize the trends in industry and academia that are in direct conflict with it and are largely responsible for the weaknesses and vulnerabilities that give hackers of all hat colors a rich ecology to exploit. A typical developer is likely to experience much of the following:

- Developers are under pressure to follow standard solutions, or the path of least resistance to "just making it work." As long as a trick works, detailed understanding is often considered optional. Consequently, they might not realize the effects of deviating from the beaten path.
- Developers tend to be implicitly trained away from exploring underlying APIs because the extra time investment rarely pays off.
- Developers often receive a limited view of the API, with few or hardly any details about its implementation.
- Developers are de facto trained to ignore or avoid infrequent border cases and might not understand their effects.
- Developers might receive explicit directions to ignore specific problems as being in other developers' domains. (I heard of a major vendor telling customers that the security of that vendor's products was the customer's responsibility. Customers were expected to "run it behind a firewall.")
- Developers often lack tools for examining the full state of the system, let alone changing it outside of the limited API.

In a typical academic setting, similar pressures exist for curriculum development. An ever-increasing number of topics limits the time students and teachers can allocate for any specific one. Consequently, instructors carefully plan their teaching environments to avoid introducing distractions (such as encountering a complicated protocol feature, an unusual software-hardware interaction, or another "border case") from the main exercise. It is common practice, for example, to create "wrapper" libraries that isolate students from unwanted complexity. Also, in OS courses, instructors frequently offset the likely time cost of interacting with real hardware by using software emulations (and the emulator is often that of simplified imaginary hardware). This can lead to unrealistic teaching environments that impart very little of the real world's actual complexity, giving students false expectations that cause problems when they join the ranks of industrial developers. (I once came across a CS introductory sequence that heavily stressed using a particular integrated development environment together with an I/O library designed to hide most of the standard system interaction and I/O complexity. Students with little independent programming experience prior to this sequence described their first internships as truly harrowing.)

Even if instructors avoid such oversimplification, they still tend to implicitly train students to follow prescribed patterns without exploring alternatives or understanding the finer details of proposed solutions. Some topics, perceived as too complicated, simply fall by the wayside (for OS courses, for example, linking and loading, binary file formats, and OS support mechanisms for debugging and tracing) and are characteristically repeated "remedially" in books that deal with computer security, despite clearly belonging in the standard curriculum.

The frustration such trends create is a driving force behind hacker culture, which eschews the path of least resistance and concentrates on fully understanding underlying standards and systems, complete with their border cases and vendor implementation differences. In particular, we can distinguish the following tendencies in the hacker methodology:

- Hackers tends to treat special and border cases of standards as essential and invest significant time in reading the appropriate documentation (which is not a good survival skill for most industrial or curricular tasks).
- Hackers insist on understanding the underlying API's implementation and exploring it to confirm the documentation's claims.
- As a matter of course, hackers second-guess the implementer's logic (this is one reason for preferring developer-addressed RFCs to other forms of documentation).
- Hackers reflect on and explore the effects of deviating from standard tutorials.
- Hackers insist on tools that let them examine the full state of the system across interface layers and modify this state, bypassing the standard development API. If such tools do not exist, developing them becomes a top priority.

These tendencies largely define how hackers learn and work, and have produced an impressive array of tools, frameworks, and exploits.

For example, most programmers must deal with linking (and, occasionally, with obscure linking errors), and every Unix distribution today relies on dynamic linking. Yet the standard CS curriculum barely covers linking mechanisms and the corresponding parts of the binary file formats; just about the only available book that

goes into sufficient depth to cover this topic is John Levine's *Linkers and Loaders*.[3] Programmers learn to interpret and fix errors, as well as to avoid situations that create them, but they usually remain in the dark about the actual mechanisms that cause them, whereas hacker publications explain these mechanisms in much technical detail (such as from *Phrack*, volumes 51, 54, 56, 59, and 61; www.phrack.org) and provide tools for examining and manipulating them, such as ELFsh (http://elfsh.asgardlabs.org). Note that although the design of the binary file format "insides" directly affects many aspects of programmers' daily activity, the knowledge of these is considered somewhat esoteric. Clearly, hackers who studied this have an advantage over typical, traditionally trained programmers.

C++ offers another example. Countless object-oriented programming books explain overloading and inheritance, both in abstract terms and on specific examples, using various pedagogical techniques. Nevertheless, students often find themselves at a loss when asked to predict the outcome of mixing overloaded and virtual functions (let alone the effects of multiple inheritance with both virtual and nonvirtual functions present). Indeed, numerous job interview puzzles have sprung up around such "trick questions." A hacker interested in the topic would likely begin his or her study by learning how overloading and virtual functions are implemented, and would discover actual compiler and linker mechanisms, such as name mangling and *vtables*,[4] after which the answers become clear, if not trivial.

Another tool ubiquitously used but rarely fully understood is the debugger. Almost all programmers have used one, yet few understand the underlying OS and hardware features that make debuggers possible. Still, only a few in-depth books on the subject are currently available (for example, the classic *How Debuggers Work: Algorithms, Data Structures, and Architecture*,[5] and Kris Kaspersky's recent *Hacker Debugging Uncovered*[6]). Yet hackers have had access to a wealth of information on the subject for a long time (for instance, on the legendary Fravia's reverse-engineering site; www.woodman.com/fravia/.) Today, the best resource of this kind is undoubtedly the OpenRCE site (http://openrce.org). Hackers had a clear advantage in this area—in fact, the industry eventually learned from them and borrowed many anti-debugging and so-called content-protection tricks (unfortunately, including such ill-advised borrowings as the "Sony rootkit").

To learn security skills, students and developers must be able to switch from their traditional conditioning to the attacker's way of thinking.

The interest in the internal workings of various programming language mechanisms is characteristic of the hacker approach. To my knowledge, a hacker is likely to learn about calling conventions and stack layouts, exception handling mechanisms such as stack unwinding and *setjmp/longjmp*, and the basics of *syscall* implementations much earlier than the average student, and often does so at the beginning of his or her own programming career. This gives hackers a set of tricks that their more traditionally trained peers might not even be aware of.

The hacker curriculum

As I pointed out earlier, various topics in a typical hacker's education are either missing from the standard CS curriculum, or are learned in a radically different fashion. In particular, tools for injecting arbitrary data (usually prepared and formatted by several layers of APIs) into the studied environment are indispensable and appear early in the hacker track. By the same token, examining the low-level state of a system ("uncooked" even by development kits) is essential. For example, when learning networking, a hacker is likely to immediately encounter tools based on the *libpcap* and *libnet* libraries for capturing and constructing raw packets respectively; hacker tutorials present a view of OS networking that is a cross-cut of the entire stack's implementation details, together with tools for affecting it on every level.[7,8]

To illustrate the hacker approach, let's look at *Phrack*, which has a long history and a strong reputation within the hacking community. Some articles that have appeared in *Phrack* are also amply quoted in academic security publications (in particular, "Smashing the Stack for Fun and Profit"[9] has become a standard reference for such attacks).

While preparing for a computer security course, I realized that most of the background material necessary for understanding modern vulnerabilities and exploits was either conspicuously absent from the traditional CS curriculum or relegated to obscure footnotes that students generally ignored. I found some of the necessary material in recently published books.[1,10] Yet, as my recommended book list grew beyond 10 items, I had to look for a different source to fill the gaps.

Phrack, dedicated to disclosing new exploitation techniques, provided an excellent selection of short articles with hands-on introductions to the missing background topics. These topics ranged from loading and relocation of binaries, dynamic linking, binary file formats, OS support for tracing and other debugging

approaches, memory allocation schemes, and IP stack implementations to common features of modern hardware rarely discussed in computer architecture courses (where they were apparently sacrificed in favor of hammering home various kinds of RISC vs. CISC arguments—consequently, it is not untypical for CS graduate students to know much about an imaginary architecture but very little about their actual desktop PC). For many years, *Phrack* has comprehensively covered emerging exploitation methods, from stack smashing, integer overflows, return-to-libraries, format string, and heap manipulation to GPS jamming and esoteric hardware faults. It covers an equally diverse spectrum of network issues.

I found that *Phrack*'s short and to-the-point introductions were often the students' first encounter with such subjects. Despite daily use of the underlying OS and network mechanisms, these topics were never a part of the students' awareness of their computing environment. From the attacker's viewpoint, this is the best of all possible worlds: the defender is not even aware of possible attack vectors, and the least-known attack vectors are the most efficient.

It is no coincidence that both *Phrack* articles and the best of recently published books on computer security fundamentals have to start with general exposition of numerous "mundane" technical topics in OSs and networking. This situation suggests that these topics are being unduly overlooked in the standard curricula. Indeed, it is hardly possible to secure systems without awareness of how they really work and how their basic mechanisms are implemented. Concentrating on only a few links in the chain is never enough for the defender.

I found that *Phrack* and similar resources also helped students develop a hacking approach to exploring or second-guessing the logic of respective implementations—a key security skill.

In my experience, the main obstacle toward developing a hacking approach was that students' attitudes, apparently conditioned by their previous programming experience, were those of developers rather than testers, reverse engineers, or attackers. Developers are rewarded for sticking to tried-and-true recipes, generally learn to trust API and interface documentation, and so on. They intentionally confine themselves to working within narrowed computing environment models for better productivity or compatibility, whereas, in reality, such confines do not exist or can be bent by the attacker.

We should not underestimate the role such conditioning plays. From an undergraduate student coding his homework assignment to a professional developer striving to meet her deadlines, programmers are under pressure to produce working, easy-to-understand code as soon as possible, leaving them no time to "question everything," explore less-used library or protocol features, or puzzle out how particular APIs are implemented (not to mention that proprietary software vendors tend to discourage the latter activity, sometimes in an extremely heavy-handed manner).

To learn security skills, students and developers must be able to switch from their traditional conditioning to the attacker's way of thinking. Exposure to the hacker culture through hacker conferences such as Defcon and others, *Phrack* and similar publications, and to comprehensive collections such as Packet Storm (http://packetstormsecurity.org) helps provide the necessary culture shock or "a-ha" moment and should be integral to every in-depth security curriculum. Recipes for preventing particular kinds of exploits are only a small part of the value these materials provide. The primary and much underappreciated value of these sources lies in facilitating a deeper understanding of the underlying systems by exposing their designers' implicit assumptions and concentrating the students' and developers' attention on the bigger picture of the system and its environment, especially on issues typically glossed over. □

References

1. J. Koziol et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley, 2004.
2. G. Conti, "Why Computer Scientists Should Attend Hacker Conferences," *Comm. ACM*, vol. 48, no. 3, 2006; www.rumint.org/gregconti/publications/20050301_CACM_HackingConferences_Conti.pdf.
3. J.R. Levine, *Linkers and Loaders*, Morgan Kaufmann, 2000.
4. *Phrack*, vol. 58, article no. 8; www.phrack.org.
5. J.B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, Wiley, 1996.
6. K. Kaspersky, *Hacker Debugging Uncovered,* A-List, 2005.
7. M. Schiffman, *Building Open Source Network Security Tools: Components and Techniques*, Wiley, 2002.
8. M. Gregg, *Hack the Stack: Using Snort and Ethereal to Master the 8 Layers of an Insecure Network*, Syngress, 2006.
9. Aleph One, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, no. 49, 1996.
10. G.E. McGraw and G. Hoglund, *Exploiting Software: How to Break Code*, Addison-Wesley, 2004.

Sergey Bratus *is a senior research associate at the Institute for Security Technology Studies at Dartmouth College. His research interests include the use of information theoretic and machine-learning techniques for security applications, as well as advances in rootkits and OS kernel security. Bratus has a PhD in mathematics from Northeastern University. He frequents hacker conventions and tries to keep abreast of the newest system and network exploitation methods. Contact him at sergey@cs.dartmouth.edu.*