Possibility Spaces: Exploits and Persuasion

In 2010, an onMouseover exploit spread through Twitter, the popular microblogging service that allows users to post 140-character tweets. The exploit—a piece of code that takes advantage of a security vulnerability—was the result of a flaw that allowed users to post executable code in Tweets using the JavaScript programming language. The spread of this exploit meant that certain users could, among other things, inadvertently repost links by hovering over certain areas of their screen. The code of this exploit is relatively simple, but to those unfamiliar with programming or with JavaScript the exploit was a curiosity. Further, the average user would likely be confused as to why anyone would create such an exploit. Most instances of the onMouseover exploit were not malicious, and even those that were did little more than propagate links to pornographic sites. No data was stolen; no passwords hacked. Still, the exploit affected the Twitter feeds of celebrities and public figures, including White House press secretary Robert Gibbs and Sarah Brown, wife of former United Kingdom prime minister Gordon Brown. Due to the exploit, Brown’s Twitter page displayed a large “h” and linked to a Japanese porn site. As the exploit hit more high-profile users, blogs and other technology publications worked to explain the exploit to readers, and one publication—The Guardian—even published a detailed account of how the hack worked, an account that I’ll examine in more detail below.

This curious exploit seems like little more than the tinkerings of a prankster, but it is actually much more than this. This hack and others like it are a direct result of a hospitable network that welcomes exploration and hacking, and they can tell us a great deal about how our digital spaces operate. My aim in this chapter is to use two different examples to demonstrate how exploits serve as ethical programs. Wikipedia provides a concise and useful definition of the term “exploit”:

An exploit (from the English verb to exploit, meaning “using something to one’s own advantage”) is a piece of software, a chunk of data, or a se-
Possibility Spaces

A sequence of commands that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized). Such behavior frequently includes things like gaining control of a computer system, allowing privilege escalation, or a denial-of-service attack.¹

Programmers find a gap and exploit it, and this once again points to the difficulties of hospitality. Short of disconnecting altogether, networked software can never remove the possibility that someone will discover a bug and exploit it. Exploits raise discussions among programmers and nonprogrammers alike about security and about ethics, and in this sense their link to rhetoric is clear. The exploit triggers conversation and has the potential to open up a space for discussions about code and software, but this is only one way of understanding the rhetoric of the exploit. While the onMouseover exploit did not articulate what we might normally call a rational, deliberative argument about web security, it did demonstrate some of the available means of persuasion. Exploits demonstrate what is possible in a given space, and they show us what is or is not available to writers and programmers. So, while the onMouseover exploit did trigger discussion about web security, Application Programming Interfaces, and third-party applications—discussions that I will cover in this chapter—its most important feature was the role it played in performing an argument about what Twitter’s website allowed users to do, thus forcing a particular feature of the Twitter ecosystem into view.

The value of this performance comes into focus when we compare the onMouseover exploit to a different, less reported Twitter exploit. Unlike the onMouseover exploit, which spread in the world of the web browser, this second exploit fell primarily in the realm of “apps”—it was a security flaw discovered in the OAuth authentication protocol that was designed to protect users of third-party applications. The OAuth protocol is part of the Twitter API (and is used by other web APIs as well). It is a security protocol that is meant to protect users of social media sites such as Flickr and Twitter who want to allow third parties access to their data (pictures, tweets, and so forth). Many of the arguments and discussions that emerged because of the onMouseover exploit revolved around the security problems with the Twitter.com interface. One blogger interpreted this incident as a reason to avoid using Twitter.com and to instead access the microblogging service via a third-party application such as TweetDeck. As networked platforms proliferate, APIs offering developers and users the capacity to build third-party applications have become more and more prevalent. They provide users and programmers with ways to access certain information and functions while also allowing companies to protect
proprietary resources. The onMouseover exploit did in fact only affect users of the Twitter.com website, while those who access Twitter’s services via third-party applications were spared the annoyance. However, the OAuth exploit is evidence that no networked space is completely safe from exploits. OAuth was initially implemented as a way to make third-party applications more secure. Prior to the institution of OAuth, users of applications such as TweetDeck, might need to provide the developers of these third-party applications access to their user credentials. The TweetDeck application would then act on behalf of the user, making requests to the Twitter API to access content. This was an insecure solution, and OAuth offered more security through a “token” system, which is discussed in more detail below. However, early in its implementation OAuth suffered its own exploit. Just as Twitter had fallen prey to a gap in its code, OAuth left itself open to an exploit that allowed programmers to gain unauthorized access to user resources.

When dealing with networked software, exploits are inevitable. Website or app, it makes no difference. Because of the hospitality built into the network, hacks will happen, but what is most instructive for our purposes is what happened in response to the discovery of the OAuth exploit. The conversation about the OAuth exploit happened among the programmers and companies that had collaborated on the design of the protocol. In fact, the existence of the exploit was kept a secret until after the problem it exposed had already been addressed. Given that the onMouseover exploit led many commentators to argue that third-party applications were more secure than the Twitter website, the OAuth exploit offers us some insight into how an exploit might emerge (and be quashed) in the world of apps. Whereas the onMouseover exploit was released on the network and raised a public discussion about how the code operated, the exploit affecting OAuth triggered discussion among a smaller group of people who were much more experienced and who had a vested interest in avoiding a public discussion.

This chapter addresses how exploits trace the edges of a digital space, demonstrating what is or is not possible. As I will argue, rhetorical theory offers ways of theorizing such possibilities. However, this requires a complex understanding of rhetoric, one that not only moves beyond popular notions of rhetoric as lying or deceiving (what Wayne Booth calls “rhetrickery”) but also beyond an Aristotelian understanding of rhetoric as a method of rational deliberation and discursive persuasion. If rhetoric is only the exchange or analysis of arguments, then it may have little force in the world of protocol, which is often concerned less with deliberations about ethics and laws than with whether or not code is operational. In addition, if we hold to a narrow reading of Aristotle’s definition of rhetoric as “the faculty of observing, in any given case, the available means of persuasion,” we might also be at a dead end
for considering rhetoric’s utility for understanding the exploit. Aristotle was interested in how the speaker or writer constructed probable truths by determining what evidence or strategies were most applicable to a situation. He was interested in what was immediately available for the task at hand, and this means that an extension into the world of the possible might land us outside of the realm of rhetoric.

My argument is that rhetorical theory has a role to play in the worlds of both “hack” and “yack” (to use a set of terms popular in digital humanities circles) because it can be productively applied beyond the space of probability and into the space of possibility. This chapter’s discussion of the fraught ethical terrain opened up by exploits will demonstrate how a rhetorical framework helps us both analyze and rewrite possibility spaces by taking up exploits as both opportune moments for analysis and as ethical programs that engage networked hospitality. More than justifying my own discipline’s relevance to software studies and networked environments, this chapter argues that exploits offer a useful pathway for understanding the rhetorical possibilities of a hospitable network. Those possibilities are exposed in a way that raises complex ethical questions. This chapter asks two questions: How do ethical programs like the onMouseover exploit and the OAuth exploit expose ethical predicaments, and how can both conversations about code and code itself respond to those predicaments?

Rhetoric and/of the Exploit

Understanding the two exploits discussed in this chapter will require that we move beyond the notion that the web is open or free. It is not my intention to argue that the “view source” option of the web browser is free from the complications of protocological control. In fact, I am less interested in evaluating the worlds of the browser and the app than I am in examining how exploits explore possibilities and raise questions in these two types of spaces. Who gets to exploit gaps in software? Who is part of the conversation when an exploit emerges? What does the exploit tell us about the software or protocols in question? These are the questions that drive this chapter. These questions are about the rhetoric and ethics of our digital spaces, about who gets to speak, who gets to code, and who gets to explore the possibilities opened up by software platforms.

But all of this is complicated by the ethics of the exploit, a practice that is less about what one should do than about what one can do in a given digital space. This is largely due to how protocological spaces work. As we learned in chapter 2, protocol operates by way of two machines, one vertical and the other horizontal. While we might be tempted to think of digital networks in
terms of the horizontal, unstructured flow of communication, a closer examination of the technologies that regulate how information moves demonstrates that this is only one half of the story. Protocological power is distributed throughout networks, exerting control by enforcing rules about how or whether information can move from node to node. These rules are enforced computationally, meaning that discussions of how they operate (or of how they should operate) will take us only so far. For Galloway, nothing is stopping us from arguing how things should happen in such networked spaces, but these arguments won’t necessarily have the greatest impact: “Opposing protocol is like opposing gravity—there is nothing that says it can’t be done, but such a pursuit is surely misguided and in the end hasn’t hurt gravity that much.” That is, the exploit operates by the maxim “more hack, less yack.”

This move toward code and away from discourse doesn’t mean that we are completely given over to rigid, unforgiving machines. Networked software establishes possibilities, but it does not necessarily do so in any final or thoroughgoing way. There is a constant push and pull between the Law of hospitality and the laws of hospitality, welcoming others to explore possibilities while also provisionally defining what those others can and cannot do.

This push and pull reminds us that though our digital spaces are programmed, they are not necessarily rigidly programmatic. Indeed, regardless of the insidious nature of protocological control, Galloway does not see it as foreclosing political action or activism. His work with Eugene Thacker in The Exploit most clearly demonstrates this and also provides us with an understanding of how rhetorical action takes shape in networks. As Galloway and Thacker note, political action in networks does coincide with a clear shift in power: “within protocological networks, political acts generally happen not by shifting power from one place to another but by exploiting power differentials already existing in the system.”

In a situation of networked hospitality, locating the origin point of power becomes difficult, if not impossible. Rather than clearly defining host and guest or locating power as a discrete thing that is exchanged between parties, the exploit becomes a rhetorical and political tactic for understanding and manipulating networked spaces. The exploit begins from the assumptions that thresholds are not clear and that the porosity of boundaries offer up the possibility of exploring multiple solutions or answers. In protocological networks, the exploit becomes a way of “discovering holes in existent technologies and projecting potential change through those holes.” These gaps serve as reminders of our constant predicament of hospitality, and acting from within this new power dynamic requires a shift in thinking, a shift that leads Galloway and Thacker to argue for thinking in terms of “possibility” rather than “probability”: 
Informatic spaces do not bow to political pressure or influence, as social spaces do. But informatic spaces do have bugs and holes, a by-product of high levels of technical complexity, which make them as vulnerable to penetration and change as would a social actor at the hands of more traditional political agitation.\(^7\)

This move away from probability, political pressure, or influence could be read as a move away from rhetoric. Rhetoric is typically understood as the study or practice of “influence” (that is, persuasion), but a closer look at Aristotle’s famous definition opens the door to a broader understanding of rhetoric. As translator George Kennedy explains, Aristotle’s definition—“an ability in each [particular] case, to see the available means of persuasion”—is very much concerned with possibility.\(^8\) Aristotle’s phrase endekhomethon pithanōn is typically translated as “the available means of persuasion,” but Kennedy explains that “endekhomenon often means ‘possible.’”\(^9\) In this sense, “availability” might be understood as casting the net widely in an attempt to see all possible persuasive resources. To be sure, Aristotle’s Rhetoric circumscribes this possibility, suggesting that rhetoric does not “theorize about each opinion . . . but about what seems true to people of a certain sort.”\(^10\) The orator does not necessarily seek “Truth” but is more concerned with what is true for a particular audience or situation. Further, the rhetor’s primary persuasive tool, the enthymeme, is concerned with probability. The enthymeme, or what Aristotle calls a “rhetorical syllogism,” is founded on the values of a community; when a rhetor builds a well-reasoned argument, she or he does so with premises that “are sometimes necessarily true but mostly true [only] for the most part.”\(^11\) These premises are within the realm of probability. So, while a speaker or writer may be concerned with the “possible” means of persuasion, she or he is primarily focused on what is probable, attempting to persuade an audience about what should be done given only partial information, what is best for a given community, or what is most likely to achieve a certain goal. Still, the role of “possibility” in the rhetor’s toolkit should not be overlooked.

While Galloway and Thacker’s account seems to suggest that persuasion would be much less useful than exploits, Aristotle’s use of endekhomenon at least suggests that rhetoric has a role to play in the exploiting of gaps and holes in networked spaces. At first glance, the idea of forcing a situation by way of an exploit might appear to be outside the realm of rhetoric, which is concerned more with discursive attempts at persuasion by way of claims and evidence and less with brute force. Rhetoric is often positioned as what we use when we don’t want to use force—a set of tools for persuading rather than exploiting. Given all of this, one might be surprised as I attempt to align the
work of the exploit with rhetoric, and so I will need to step back to go for-
ward, making clear the relationship between the exploit’s interest in possi-
bility (what is or is not possible in a given digital space) and rhetoric’s interest in persuasion and influence.\textsuperscript{12} I offer this all too brief detour to set the scene for my own argument about how rhetoric intersects with the exploit and also to deepen software studies’ relationship with rhetoric. As we saw in the previous chapter, Bogost’s procedural rhetoric plays a part in my own theorizations of rhetoric and software. However, the remainder of this book will demonstrate the broader usefulness of rhetorical theory for the study of software and computation.

Dilip Gaonkar offers the most succinct description of how rhetoricians have wrestled with probability, possibility, and contingency.\textsuperscript{13} Gaonkar argues that Aristotle’s focus on probability and contingency in his \textit{Rhetoric} was an attempt to rescue rhetoric from Plato’s famous critique that it was flawed in “its reliance on appearance, its entanglement with opinion, and its linguis-
tic opportunism.”\textsuperscript{14} By placing rhetoric in the category of the contingent and thus opposing it to necessity—rhetoric deals not with what must be true but rather with what could be true—Aristotle opens up space for rhetoric outside of philosophy, and his \textit{Rhetoric} theorizes that space by offering a method for deliberation:

\begin{quote}
If human beings can act in more than one way (and if the outcome of their actions is uncertain, capable of unanticipated consequences), then it makes sense to deliberate and choose. Rhetoric is the discursive medium of deliberating and choosing, especially in the public sphere.\textsuperscript{15}
\end{quote}

For Gaonkar, Aristotle’s description of rhetoric in terms of probability is a way to “domesticate” it. Aristotle is not interested in pure contingency but rather in what is \textit{probable} within the space of the contingent. Gaonkar describes this approach aptly, suggesting that Aristotle is not interested in a “Kafkaesque world of sheer uncertainty and terror but rather a world made familiar by Emily Post—of gamesmanship and good manners displayed by those adept at ideological bricolage.”\textsuperscript{16} Thus, Aristotle finds a safe and stable home for rhet-
oric in contingency and probability, one that can be systematically theorized.

But the space of probability and contingency might rein things in too much, making rhetoric’s purview too neat and tidy. Even if Gaonkar has little interest in expanding or “sizing up” rhetoric, he does suggest that the “unyok-
ing of the contingent from the probable, if rendered explicit and thematized in future studies, might produce new and challenging possibilities in our un-
derstanding of rhetoric.”\textsuperscript{17} One such effort is carried out by Nancy Struever in \textit{Rhetoric, Modality, Modernity}, and while Struever’s text never touches upon
Possibility Spaces

Software or networked life, her unyoking of the contingent from the probable offers us a way to theorize rhetoric alongside the exploit. Struever argues that rhetoric’s promise lies not in a focus on probability but rather in its openness to multiple paths of thought. The details of Struever’s analysis of Thomas Hobbes, Giambattista Vico, and Walter Benjamin and her concern for the relationship between rhetoric and philosophy are outside the scope of this project, but they are at least worthy of mention here. Struever convincingly argues that Hobbes, Vico, and Benjamin provide valuable resources for thinking through contemporary civic inquiry, but the value of these thinkers is more in their penchant for possibility (and their sometimes unwitting turn to rhetoric) rather than in philosophy’s traditional terrain of necessity. In these thinkers, she sees “modal rhetorics” that are more attuned to the problems of modernity (and, we might argue, postmodernity) than is much of the Western philosophical tradition. But most important, for our purposes, is Struever’s argument that a rhetorical approach is “non-dismissive”:

There is a deep compatibility between the very specific analytic techniques rhetoric must develop to fulfill the demands of persuasion, the core political functions, and the very general commitment to the modality of possibility as the domain of rhetorical duty. There is a beneficial interactivity of modal proclivity and analytic habits that energises; it is profoundly non-dismissive.¹⁸

For Struever, rhetoric is defined by its mode of possibility, its continuing hospitality to approaches to problems. This is an ethical stance, one that welcomes any attempt as a possible solution, even if it is initially unclear what that attempt will yield. While Gaonkar’s account of the Aristotelian tradition shows a persistent link between contingency and probability among theorists of rhetoric, Struever pries these two terms apart, radically expanding rhetoric’s possibilities. The focus on possibility means that Struever’s rhetorician aims to explore all avenues, follow all pathways, and trace the edges of a given rhetorical space. Rhetorical tactics demonstrate these possibilities, regardless of whether or not they appear logical at first glance. This profound nondismissiveness is also what defines Galloway’s notion of the exploit, which sometimes proceeds regardless of a clearly considered ethical program. Just as Galloway and Thacker’s exploit seeks out “power differentials already existing in the system,” Struever’s understanding of rhetoric emphasizes its “remarkable capacity for renewal, for ‘modernizing,’ the reinvention of its civil strategies in response to novel civil affairs.”¹⁹ Struever’s focus is on the challenges of modernity writ large as she demonstrates how rhetoric has been renewed from early modernity
forward. This project of renewal is all the more necessary as we are faced with novel rhetorical problems in computational environments. The rhetorician of networked, computational environments must be nondismissive, seeking out the exploits that pry open gaps and demonstrate possibilities. It should be noted here that the world of the exploit is not a rosy one, and we’ll see examples of this in the analysis below. I’ll return to the troubling ethical terrain of the exploit at the end of this chapter.

Struver’s insistence that rhetoric is “non-dismissive” suggests that rhetoric and the exploit share a radical openness to possibility. The exploit is rooted in profound nondissimissal and is pure possibility. It is not always a rational or well-considered attempt to institute a political or ethical program, and it is many times not even sure of its own direction. It is about exploration. Again, the parallel between rhetoric’s nondissimissiveness and the exploit is striking. As Galloway and Thacker suggest, the exploit lays the groundwork for political (and, I would add, rhetorical) action: “look for traces of exploits, and you will find political practices.”

While these exploits are not always so obviously pragmatic, they are also not always malicious:

Contrary to popular opinion, not all computer viruses are destructive (the same can be said in biology, as well). Certainly computer viruses can delete data, but they can also be performative (e.g., demonstrating a security violation), exploratory (e.g., gaining access), or based on disturbance rather than destruction (e.g., rerouting network traffic, clogging network bandwidth).

The exploit is profoundly rhetorical in its search for possibilities and its desire to demonstrate and perform those possibilities. Anyone who has been the victim of “clickjacking” on Facebook is aware of hacks that are more about performance and possibility than about stealing login information. In June 2010, many Facebook users following links that read “Justin Biebers [sic] Phone Number Leaked!” were taken to a page with another link that read “Click here to continue if you are 18 years of age or above.” Clicking anywhere on this page “launches an invisible iframe [an HTML element that contains another HTML document] which contains a Facebook like button, thus spreading the link to more and more users.” If you fell for the Bieber clickjacking exploit, your friends discovered that you were at least mildly interested in Justin Bieber’s address and telephone number. But this embarrassment (which may or may not be mortifying, depending on the victim) is the extent of the damage. In this particular case, the clickjacking “attack” contained no real threat and stole no personal data. This kind of hack is more about doing what is possible than about gaining access to data or resources, and it is here that the
intersection between hacking and rhetoric makes the most sense. But even if such hacks are not overtly political and even if they seem to be little more than annoyances, they do, as Galloway and Thacker suggest, point the way to political practices. They also expose ethical predicaments. The exploit, in its search for endless possibilities and in its willingness to accept the welcoming gesture of the hospitable network, is a powerful (even if sometimes troubling) ethical program.

The onMouseover Exploit

In August 2010, Masato Kinugawa noticed a vulnerability that would allow a Twitter user to post JavaScript into a tweet. This security flaw left Twitter open to an XSS worm, “a malicious (or sometimes non-malicious) payload, usually written in JavaScript, that propagates among visitors of a website in the attempt to progressively infect other visitors” (Wikipedia, “XSS Worm”). A user could post lines of JavaScript code as a tweet, which would then execute on any machine used to view that tweet via the Twitter Web page. The flaw was actually rooted in Twitter’s decision to allow the “@” symbol in URLs posted in tweets. We can see this by looking at the Twitter text-processing library, the documentation that explains how Twitter’s software converts a URL into a clickable link. A website called Stackoverflow.com, a question-and-answer site for programming issues, points us to the problem code:

```python
60 # Allow @ in a url, but only in the middle. Catch things like http://example.com/@user
61 REGEXEN[:valid_url_path_chars] = /(?:
62 #REGEXEN[:wikipedia_disambiguation]]
63 @[^/]+/\|\|
64 \[.\.,]?#{REGEXEN[:valid_general_url_path_chars]}
65 )/ix
66 # Valid end-of-path characters (so /foo. does not gobble the period).
```

A post to Stackoverflow.com by a user named Brian McKenna explains the “offending regex”: “The `@[^/]+/` part [line 63 in the code above] allowed any character (except a forward slash) when it was prefixed by an @ sign and suffixed by a forward slash.” Regex stands for “regular expression,” and it defines how certain strings of characters should be matched to other strings of characters. In this case, the regex is controlling how a URL entered by a user is converted into a clickable link. While investigating the regex, Kinugawa noticed that a URL ending with the character string @” would trick the Twitter parser into thinking that it had received a valid URL and would also allow
someone to include JavaScript in a tweet. Another contributor to Stackoverflow.com explains the flaw this way:

Here’s an example of a link one could post in a tweet prior to this flaw being patched:

```
http://thisisatest.com/@"onMouseover="alert(‘test xss’)”/
```

Twitter’s software notices the link and puts it in an href tag:

```
<a href="http://thisisatest.com/@"onMouseover="alert(‘test xss’)” rel=""
target="_blank"="">http://thisisatest.
com/@"onMouseover="alert(‘test xss’)”</a></span>
```

Twitter’s parser sees `http://` and a trailing slash (the “/” character at the very end of the tweet) and assumes that this is a valid URL. Because the above regex allows any set of characters in between the @ symbol and the trailing slash (except for a forward slash), the text between the @ and the / is a virtual playground for the JavaScript hacker.

The onMouseover portion of this code is what caused problems for people such as Robert Gibbs and Sarah Brown. When users dragged their mouse over this tweet, any number of events could be triggered: users could be redirected to other web pages, pop-up windows could appear, or the user’s own account could be used to “retweet” (Twitter lingo for passing along another user’s tweet) links to any website. Many who used this exploit, including Kinugawa, used colored blocks to entice users to mouse over this section of the tweet (see figure 3).

Twitter eventually fixed this problem by changing the regex so that only valid URL characters were allowed. Thus, when quotation marks are included, they are now converted into the character string `quot;` (the HTML entity for a quotation mark). This conversion prevents quotation marks from acting as they would in JavaScript code, but it doesn’t stop the user from entering the quotation marks into a tweet. Nothing stops someone from attempting this same exploit, but that attempt no longer allows the user to create executable JavaScript. One could still enter the same string of text that caused problems previously, but the Twitter parser no longer allows for executable JavaScript code. This is an important example of protocol at work. Rather than stopping users from typing the string of characters that caused the problem (policing the behavior that caused the problem), this change to the regex merely neutralized the power of that string of characters, preventing it from executing code (policing the effects of the behavior).

Upon noticing this problem in August, Kinugawa claimed that he noti-
fied Twitter, and Twitter claims that it did in fact fix the problem with the regex. However, Kinugawa noticed the problem again in September, and this is when he experimented with the exploit. Twitter’s blog explains that the flaw was inadvertently reintroduced sometime between August and September: “We discovered and patched this issue last month. However, a recent site update (unrelated to new Twitter) unknowingly resurfaced it.” The mention of “new Twitter” is important here, as Twitter’s damage control was not only about a technical flaw but also about marketing their newly designed web page. Around the same time the onMouseover exploit caused these problems, Twitter had rolled out a newly designed web interface. This blog post insists that the flaw had nothing to do with that new design, and there is no reason to think otherwise. But more important, Kinugawa’s exchanges with Twitter demonstrate the difference between pointing out the problem and exploiting it. Notifying Twitter of the problem did in fact trigger a fix, but that fix was inadvertently undone. Kinugawa, assuming that Twitter had failed to address the problem, chose a different rhetorical tactic, demonstrating the problem by way of the onMouseover exploit. Rather than pointing to the problem (a strategy that seemed to fall short), he used code to demonstrate what could be done when such a flaw was present. Kinugawa’s exploit stands as an ethical program that demonstrated what was possible in this computational space. By exposing the problem—and not necessarily solving it—the exploit triggered a great deal of public debate. That is, it was rhetorical twice over. It
demonstrated the possible means of persuasion in this computational, net-
worked space while also triggering a wide-ranging discussion about web se-
curity, cross-site scripting attacks, and third-party applications.

While Stackoverflow.com provided a detailed discussion of how the ex-
plot worked and opened up discussions about the implications of the exploit, news outlets also tried their hand at explaining the incident. Gawker, better
known for its snark than its technology reporting, provided a concise expla-
nation of what happened:

Kinugawa’s techniques, now shown in the wild, were rapidly picked up
by others. What he’d discovered was that Twitter failed to properly filter
tweets for Javascript code. Or rather, it did filter out Javascript, unless your
URL contained the “@” symbol, in which case you could trick Twitter into
accepting your Javascript in a tweet, and then embedding that Javascript
when it displayed your tweet to other users. This sort of attack, known as
“cross-site scripting,” or “XSS” for short, is a classic and well understood
phenomenon that Web developers are routinely badgered to be on guard
against.29

Reporting like this provided readers with an explanation of the code rather
than merely providing a description of the exploit’s fallout. This is at least
some evidence of a public that is interested not only in being told that a soft-
ware problem exists but also interested in understanding how that software
works.

But the most comprehensive coverage seems to have come from Guardian
reporter Charles Arthur, who teamed up with software developer Richard Gay-
wood to provide an explanation of how the hack worked. The Guardian’s cov-
rage is worth citing as a way of understanding the onMouseover exploit, but
also important is the very fact that this major news publication spent time on
an explanation of the actual code. As Noah Wardrip-Fruin argues in Expressive
Processing, such detailed and accurate journalism is crucial if the public is to
gain a meaningful understanding of the cultural, ethical, and expressive im-
lications of computation. In his discussion of Selmer Bringsjord and David
Ferrucci’s Brutus story-generation system, Wardrip-Fruin notes that press
coverage of the system described it as “a story author.” Wardrip-Fruin pro-
vides a painstaking account of the system’s operations, showing us that the
stories generated by Brutus are, by and large, not generated by the system’s
procedures and operations. Instead, the operational logic of Brutus is “that of
a child’s picture puzzle. Each piece can only fit in one place, in a manner deter-
mined by the authors before the system is set running.”30 Thus, Brutus wasn’t
authoring stories. It was arranging story content that had been authored by
Bringsjord and Ferrucci. But this didn’t stop the New York Times from suggesting that Brutus could “spit out its story in seconds.” Wardrip-Fruin points to such press accounts (and corrects them) in the hopes that software studies can “become pervasive enough that, in the future, those writing for the media will be less easily fooled.” The detailed account of the onMouseover exploit published by The Guardian offers some hope in this regard.

Gaywood explains the code as “a classic piece of Javascript injection,” and he provides much the same explanation as the Stackoverflow.com contributor. He includes an example of a tweet using the XSS code that had caused so much havoc and explains how Twitter’s software would handle it. He correctly diagnoses the problem, explaining that the @” text string “broke their parser” and allowed people to embed JavaScript in their tweet. He also explains that Twitter did not follow the knee-jerk reactions of those who suggested blocking users from including the string “onMouseover” in tweets: “Twitter fixed this not by blocking the string onMouseover (which some dim-witted blogs were calling for) but by properly sanitising the input. The “quotation marks in these tweets are now turned into &quot;—the HTML-escaped form.” Blocking the text string “onMouseover” is a “dim-witted” response because it does not get at the root of the actual problem—it treats the symptom but not the cause. This would solve the problem of this particular hack, but it does not address the actual gap opened up by the flaw in the Twitter regex. It would enact a short-sighted ethical program, without accounting for the deeper problem exposed by Kinugawa’s exploit. By converting quotation marks to their HTML-escaped form, Twitter correctly solves the problem at its root, even if, as Gaywood explains, such a simplistic attack is “rather embarrassing for Twitter.” This coverage by The Guardian is one of the more detailed accounts of what happened, although blogs and websites such as Wired, PC Magazine, and Lifehacker covered the story as well (I address the coverage of these publications below). This discussion of code in language that most average readers can understand provides some hope that Wardrip-Fruin’s concerns are being addressed by certain publications. In fact, readers praised the coverage in the comments section and Arthur answered critiques of the coverage by joining the blog comment conversation.

But this discussion about software—a rhetorical exchange about code—is triggered by a rhetorical action with code, an exploit that demonstrated its argument instead of explaining it and used computation as a rhetorical medium. In effect, Kinugawa’s little bit of code finds a small (and, as many argued, embarrassingly simple to fix) gap in Twitter’s architecture and exploits it. It’s not clear whether users saw Kinugawa’s exploit and copied it, or whether they discovered the flaw in Twitter’s regex independently (though, the latter seems unlikely given how quickly the XSS script spread throughout
the day). Regardless, the XSS worm took on various forms. While Kinugawa demonstrated how to create colorful blocks of text in a tweet, Magnus Holm extended the exploit so that users would retweet and spread “infected” tweets by mousing over certain parts of the screen.34 This hack is arguably the one that was most jarring to average users of Twitter (in particular those who had no experience with JavaScript). While the average user may in fact know that mousing over an image will trigger certain kinds of events (for instance, displaying a caption or a pop-up window), the norms of web design have meant that very few users expect the onMouseover event to lead them to a different page or to trigger a retweet from their own account. This is one more instance of the onMouseover exploit gesturing toward the possible and alerting users to how software works. The exploit moves beyond norms and generally accepted design principles (moving away from arguments about decorum) in order to perform the possible (moving toward the execution of ethical programs that demonstrate what can or cannot happen in a given space).

Reshaping the Possibility Space

When interviewed, Holm insisted that that he “simply wanted to exploit the hole without doing any ‘real’ harm.”35 Another user who explored this security flaw was Pearce Delphin, a 17-year-old Australian. As with Holm, Delphin insisted that this was about discovering a vulnerability. In their piece on the exploit, the website Mashable.com included some finger-wagging for Delphin:

[Delphin] hopes he won’t get into trouble, but he very well could—the proper course of action in situations like these is reporting such a vulnerability to Twitter. Exposing a security flaw like he did, even inadvertently, is at the very least an error in judgment.36

This tsk- tsk regarding the “the proper course of action” and the accompanying mention of a lapse in judgment points to the difficult ethics of the exploit. Kinugawa claims to have initially followed the “proper course of action,” but after seeing that the fix had not been implemented (or that the fix had not remained in place) he chose a different course of action. But to simply blame Delphin, Kinugawa, or Holm for a lapse in judgment is to misunderstand the ethics of the network and the ethical programs that institute our digital spaces. Moralism and recommendations about what should have been done by these hackers is not a particularly useful response to this situation. Exploits tell us something about the software, about the network, and about what is possible in this space. Further, one would have to admit that the exploit was more rhetorically effective in ensuring that this particular gap was addressed. Yack did not result in a solution. Hack did.
But in addition to fixing this particular problem with Twitter’s parser, the onMouseover exploit is a useful example of how public discussions about code can help educate programmers and nonprogrammers alike. The exploit opened up a discussion about how Twitter handles URLs and how its parser works. Thankfully, news outlets such as The Guardian presented us with an accurate and carefully considered description of the problem. This explanation of code provided the public with a deeper understanding of how this digital space works. Assigning responsibility is difficult when thinking through the ethics of the exploit. Who is responsible? Twitter, for not patching this hole in any thoroughgoing way? Kinugawa, who initially (by his own account) attempted to aid Twitter in patching the flaw? Delphin and Holm, who were exploring the possibilities opened up by Kinugawa? Despite the search for an origin story, the onMouseover exploit forces us to confront difficult questions of responsibility. However, it provides us with no clear answers. Given this predicament, it seems more fruitful to understand the ethical questions being negotiated by way of both code and language than it does to search for a blameworthy subject.

Responsibility in such situations is difficult, and digital spaces are not a free-for-all in which anything goes. But what can happen (the possibility space) in a given networked environment is largely defined by the ethical programs that shape it, and those programs accept the invitation extended by the hospitable network. Oftentimes this means that the possible is continually redefined by the ethics encoded into the software. Who or what can or cannot arrive in a digital space? Who writes the ethical programs of a digital environment? Who writes the code that stands as gatekeeper, simultaneously welcoming and patching up the exploit? When the actions of Holm are characterized as a lapse in judgment, the assumption is that he has breached the ethical code of this space. But the software would suggest otherwise. Twitter is essentially useless without the hacking and exploration of users like Holm. From hashtags (labels that allow users to aggregate similar tweets) to the third-party applications developed to aid victims of natural disasters, Twitter relies upon users and programmers for its very existence. This space invites hackers and developers to explore, but it does so only to a certain point, and this is what the onMouseover exploit truly exposes. The exploration of the possibilities of Twitter—in other words, the rhetorical tinkering of users in this space—reaches its limit when a hack is disruptive. This should not surprise us, and we would never expect Twitter to welcome all exploits, malicious or otherwise. But recognizing these limit cases as ethical programs allows us to carefully consider how the laws of hospitality are continually in conversation with the Law of hospitality.

One might ask: What does an exploit like this one transform? It seems to be little more than a playful attempt to hijack Twitter feeds and star-
tle users. However, this prank became a justification for a profound shift in our software landscape that is already under way. A minor glitch, allowing an @ symbol at the end of a URL, set in motion not only an XSS worm but also a conversation about software. The onMouseover exploit was about exploring the possibilities opened up by that minor glitch, but the response to the exploit and its aftermath led many to argue that third-party applications were preferable to websites. Many publications used the onMouseover exploit as evidence that the third-party applications (or even Twitter’s own mobile applications) are more secure than the Twitter.com website. In the aftermath of the onMouseover exploit, PC Magazine suggested that users who hadn’t already moved to a non-HTML based client should consider doing so:

One good defense is to use a third-party, non-HTML based Twitter client. Some could be vulnerable, but it’s less likely. The solution for Twitter, filtering out JavaScript, should be relatively straightforward and may be applied by the time you read this.37

The technology blog Lifehacker gave the same diagnosis and advice:

The exploit has spread to thousands of accounts now—some with hard-core porn pop-ups, other with jokey references to the exploit—so stick with a third-party Twitter client for the time being to read and send your short updates.38

But while PC Magazine and Lifehacker seem to suggest third-party applications as a short-term solution, Wired author Tim Carmody went so far as to say that this episode was evidence that the desktop browser was good for only one thing:

[The mouseover exploit] reinforces my longstanding belief that web browsers’ only legitimate use on the desktop is for viewing and watching porn (including, naturally, technology-and-gadget porn, like what you find here at Wired.com–TC); client applications, whether on a personal computer or a mobile device, are ideally suited for consuming and exchanging information.39

Along these same lines, Gawker also noted that people shouldn’t be accessing tweets via Twitter.com, and that the exploit should be seen as a teaching moment:
One lesson, though, is clear: It’s absolutely safer to have people reading tweets through a diverse array of software products than through a single website. Given that Twitter is trying hard to draw people back in to Twitter.com, where it can show them large-format ads featuring images and video, it must regret serving them up such a harsh lesson in the danger of trusting a single company to be the hub for so much information.\(^\text{40}\)

Most agreed that the onMouseover exploit was one more reason to shift away from websites and toward third-party apps. This exploit did not cause the shift from the web to apps, but it was certainly used by many as justification for sliding away from the world of web pages and toward the world of apps. This conversation about the possibility space of Twitter turned into a conversation about avoiding such problems altogether. Rather than entering a space that extends an invitation to the exploit, a space that is open to these kinds of tinkerings, many recommended that the world of APIs and third-party applications was a more secure and pleasant digital space.

As we will see, the worlds of apps and APIs are not free of exploits. How those exploits are addressed points to the difference between websites and apps. But it is worth pausing to consider what this move toward apps and APIs means when considering the possibility of, to use Annette Vee’s term, a procedurate public. This move would mean that a broader swath of people, should they desire to gain a deeper understanding of software, would have to learn the ins and outs of APIs. APIs provide a threshold between software companies and outside software developers—they are the filtering mechanisms that allow companies to provide those outside the company with the resources to extend the value of the company’s software. Facebook, Google, Twitter, and a host of other companies have APIs, and these APIs are their primary answer to the predicament of hospitality.

The push to open up software has been answered by the API, but this response to hospitality does not merely throw open the front door. Instead, it stands as an instance of the laws of hospitality, simultaneously inviting and filtering those seeking access to software and data resources. One could easily argue that the API provides programmers with more access to software and code than ever before. The API offers a middle way between proprietary, guarded software and open source software. However, that middle way still carries with it a definitive split between those with programming know-how and those without it. We can examine the possibility space of APIs and apps by comparing the onMouseover exploit, which triggered a broad conversation, to an exploit that resulted in a conversation among a decidedly different group of people.
OAuth: A Different Conversation about Code

As the footprints of social networking sites grow, the importance of web APIs grows as well. APIs allow developers to request data from a service such as Twitter, and applications designed with APIs often make these requests on behalf of software users. These requests are API calls, and they require that the application prove that it has permission to access certain resources. This permission is regulated through the use of certain authentication protocols—rules that the software must follow in order to gain access to information. In August 2010 (around the same time that Kinugawa noticed the flaw in Twitter’s URL parser), Twitter instituted a significant change and turned off basic access authentication (Basic Auth). This forced all API calls to use the Open Authorization (OAuth) authentication protocol, meaning that a user’s credentials could no longer be sent along with the API request. Under Basic Auth, API requests include the username and password. So, when an application—say a third-party Twitter application—wanted access to a user’s resources (their tweets) in order to display them or, in the case of one application, in order to calculate a user’s “Tweet Stats,” it would have to authenticate with Twitter’s servers by including that user’s credentials. This protocol was flawed for a number of reasons. It forced users to hand over login information, made the application useless if the user decided to change passwords at a later date, and gave users no way to easily track which applications had access to their credentials. As more and more developers began to design third-party applications (and as more and more malicious applications began to abuse access to user credentials), Twitter realized that it needed a more secure authorization protocol. Twitter was not the only company looking for a solution to this problem. Google, Facebook, and other companies with popular APIs also needed a more secure way for users to allow access to data.

Enter OAuth, which Eran Hammar-Lahav, one of the architects of the protocol and eventually one of its more vocal critics, describes as a “valet key” for the web:

Many luxury cars come with a valet key. It is a special key you give the parking attendant and unlike your regular key, will only allow the car to be driven a short distance while blocking access to the trunk and the onboard cell phone. Regardless of the restrictions the valet key imposes, the idea is very clever. You give someone limited access to your car with a special key, while using another key to unlock everything else.

As the web grows, more and more sites rely on distributed services and cloud computing: a photo lab printing your Flickr photos, a social net-
work using your Google address book to look for friends, or a third-party application utilizing APIs from multiple services.

The problem is, in order for these applications to access user data on other sites, they ask for usernames and passwords. Not only does this require exposing user passwords to someone else—often the same passwords used for online banking and other sites—it also provides these application[s] unlimited access to do as they wish. They can do anything, including changing the passwords and lock users out.

OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.).

Whereas Basic Auth asked developers to send user credentials along with an API request, OAuth provides users with the option of approving access to their resources. Further, it allows users to grant access to certain resources while protecting others. This is why the valet key metaphor works so well. Users are no longer required to provide access to all data. Instead, they can provide limited access.

The change from Basic Auth to OAuth is a significant one that adds a layer of complexity to the API call. For instance, under Basic Auth, a user could use his or her own credentials to access any other user’s Twitter timeline. This functionality is lost with the move to OAuth. Under Basic Auth, the following command would allow developers to access a Twitter timeline:

```
%curl -u jamesjbrownjr:jimstwitterpassword http://twitter.com/statuses/user_timeline.rss
```

Notice that this command sends the username and password along with the request. Typing this command today triggers the following error code:

```<error code="53">Basic authentication is not supported</error>```

This is understandable, since sharing actual usernames and passwords is not the most secure way of dealing with user credentials. In the case of a third-party application, it means that the application making the request is “pretending to be the resource owner.” If the resource owner (the user who is granting an application access to his or her information) has handed over her credentials, then this transaction is aboveboard. I hand over my credentials, someone/something acts on my behalf, and I do this so that I can gain the functionality of a useful third-party application. But given that my credentials
can easily fall into the wrong hands, a more secure protocol would need to find a way for me to grant access to my resources without revealing my username and password. This is what OAuth allows users and developers to do. Rather than providing credentials, the user instructs Twitter to grant access to his or her resources via a token credential:

Token credentials are used in place of the resource owner’s username and password. Instead of having the resource owner [the Twitter user] share its credentials with the client, it authorizes the server [Twitter] to issue a special class of credentials to the client [the third-party application] which represent the access grant given to the client by the resource owner. The client uses the token credentials to access the protected resource without having to know the resource owner’s password.44

OAuth allows users to grant access to their data without granting access to their credentials.

OAuth is an open protocol. No single entity owns it, and any number of people and companies had a hand in designing it. This means that OAuth is the result of a massive collaboration. However, this does not mean that all parties to that collaboration had an equal voice or an equal stake in determining how the protocol should work. One contentious moment in OAuth’s development involved the requirement of “signatures” as a way of ensuring that user credentials were secure. Rather than transmitting a password over a network and relying on that network to employ secure protocols, the use of signatures would mean that a secret (a password) would be “used to calculate a value which cannot be converted back [into] the secret itself.”45 The secret value is never transmitted over the network, protecting users from situations when secure protocols such as Secure Sockets Layer (SSL) have not been implemented. Signatures ensure that secrets are not sent “on the wire,” but they require some extra effort on the part of those designing APIs. The difficulty of implementing signatures led large companies to bristle, and signatures were removed in the move from OAuth 1.0 to OAuth 2.0. This meant that the token credentials were still used but that the “secret” behind that token was sent through the network without any encryption. Thus, this solution was not as secure as it could have been.

Hammar-Lahav argued for the use of signatures, suggesting that companies such as Twitter, Google, and Microsoft should create useful libraries for developers and make implementation of signatures easier. This would mean that developers would not have to code things from the ground up each time they needed to implement signatures. Instead, developers could rely on a set of pre-
fabricated functions. However, instead of building such libraries and working through the difficulties of implementing a more secure solution, Google and Microsoft proposed their own protocol called WRAP. Instead of using signatures, WRAP relied on cookie technology for security. In response to this proposal, Hammar-Lahav asked (sarcastically): “Why bother to create something more secure if it makes it harder for developers to use, while not actually improving the overall security of the service[?]” Hammar-Lahav served as the primary editor of the OAuth protocol. So, when WRAP began to siphon off interest in OAuth, he was concerned. While he did not accuse the creators of WRAP of attempting to replace OAuth, he did realize that the creation of WRAP meant that the battle over signatures was all but over: “At the end, due to internal corporate politics and product release schedule, Microsoft and Google decided to ship WRAP implementations and positioned it as a complete solution available as an OAuth replacement.” Ultimately, Hammar-Lahav recognized that the move to a signature-less protocol was inevitable, largely because powerful companies were leading the charge: “The bottom line is that the OAuth community who created the original specification doesn’t exist anymore. Instead, we have a few individuals who carry enough recognition to make others follow them as if they represent a community.”

Eventually, WRAP was deprecated, but not before the functions that the WRAP creators were pushing for had been incorporated into OAuth. Thus, OAuth 2.0 was released without requiring signatures, something that Hammar-Lahav believed was a big mistake. OAuth 2.0 was, for Hammar-Lahav, a step backward. He was not alone. Mozilla software engineer Ben Adida argues that OAuth 2.0 “might actually be worse than passwords.” It uses tokens, but those tokens are sent directly over the channel. If the channel is not secure—if, in Adida’s words, someone “forgets to turn on SSL”—the token is left in the open for all to see. This means that users are at the mercy of web developers:

[A]t least you can work to educate users about SSL (and after their Facebook account gets hacked, they might actually care), but it’s very hard for users to gauge whether web applications are doing the right thing with respect to SSL certs when the SSL calls are all made by the backend which has trouble surfaced certificate errors.

OAuth puts security in the hands of individual developers, and if those developers make a mistake there is little the user can do (short of not using the application anymore). Adida, like Hammar-Lahav, recognizes that good web security requires effort:
I understand. Security is hard. Getting those timestamps and nonces right, making sure you’ve got the right HMAC algorithm... it’s non-trivial, and it slows down development. But those things are there for a reason. The timestamp and nonce prevent replay attacks. The signature prevents repurposing the request for something else entirely. That we would introduce a token-as-password web security protocol in 2010 is somewhat mind-boggling.\textsuperscript{50}

Doing things the right way would have certainly been more difficult, but the WRAP compromise was an easy fix that compromised security. Despite all of this, Hammar-Lahav recognized that open protocols require compromise, and he still remained a central voice in the project.

The conversation about OAuth happened among software developers, corporations, and anyone else with the know-how and desire to get involved. In theory, anyone could be part of this conversation. In practice, those who are plugged into such issues, who attend conferences on software design and web security (many meetings concerning OAuth happened at conferences), and who wield the most power will determine how this particular ethical program is authored. This means that a piece of software that serves as a key player in determining how users and software navigate the hospitable network was designed and implemented by a small circle of people and companies and that the resulting ethical program answers to the needs and desires of that circle. Regardless of how open the conversation about this protocol was—or, perhaps because it was so open—certain voices and interests were able to drive the writing of the protocol. Given the clout of Google and Microsoft and the ease with which WRAP dealt with security, it did not matter that OAuth 1.0 was most likely the more secure solution. Many (including Adida and Hammar-Lahav) agreed that OAuth 1.0 was too complicated and needed to be reworked.

We can see that this conversation was conducted very differently than the conversation surrounding the onMouseover exploit. When Kinugawa’s exploit emerged it resulted in a conversation among a fairly broad range of people. The conversation about OAuth was much more limited, and while a close look at how OAuth deals with signatures provides some important insight into how software and protocols are built, a more apt comparison between the world of the web and the world of apps would require looking at how a particular exploit was dealt with by the OAuth community. This is where we turn next.

OAuth Exploit

In April 2009, a security flaw was discovered in OAuth (the exploit did not have anything to do with signatures). Following how the exploit emerged
Possibility Spaces

...and how it was addressed can help us understand the difference between the ethical programs enacted in the wild and those that emerge in the more cloistered world of software developers and corporations. The world of apps is not closed, and the world of the web is not open. However, the two are very different, and the OAuth exploit triggered a very different set of events compared to the onMouseover exploit.

Twitter had recently rolled out its OAuth authentication service, and it had announced that Basic Auth would soon be disabled. However, when they shut OAuth down in April without any explanation, many complained. Twitter provided no explanation for its decision, something that became understandable once developers (and the press) discovered what was happening behind the scenes. We now know that the shutdown happened because a hacker had discovered an exploit. Scott Loganbill of Webmonkey.com explains the exploit this way:

Determined to find an exploit, the hacker (who prefers to remain unnamed due to the terms of his employment) targeted OAuth. The hacker found that if he started a request, then directed a victim to initiate the authorization form on his behalf from a bogus trap site, the victim would submit the login form and provide the hacker access to the victim’s data.51

Like Kinugawa, this unnamed hacker used the exploit to perform (and not merely point out) a flaw in the protocol. Those designing and implementing the protocol now had a significant security problem on their hands.

OAuth’s normal flow of events goes like this: a user visits a third-party site, that user approves access to their resources, and he or she is then redirected to an authentication page in order to enter login information. At this point, the third-party receives a token allowing that third party access to the user’s resources. What this programmer discovered was that this entire process could be stopped in the middle, that the third party could take note of the token information and then direct users to a “trap site” on which the user would grant access to his or her account. A hacker taking advantage of this exploit never actually has access to the user’s credentials, but that hacker now has access to the user’s resources (pictures, Tweets, and so forth). This is called a “Session Fixation” attack, and Marshall Kirkpatrick of Read Write Web explains it succinctly: “The problem arose if an attacker could convince you to complete their request for account permission with your login. At the end of the process they would have access to your account.”52 The exploit takes advantage of the ability to pause the flow of information and credentials midstream.

As Loganbill explains, this exploit was discovered early on, and most stakeholders were present at a conference called Foo Camp when the exploit emerged:
The good news is the exploit was found before it was used on any other use case than Twitter. The bad news is that once the exploit was discovered, OAuth experts realized other OAuth partners weren’t safe either. Because around 75% of OAuth adopters were gathered at Foo Camp by luck, the primary shareholders all agreed on a course of action to take to minimize damage.53

That course of action involved Twitter shutting off its OAuth service without explanation, since it was Twitter’s implementation that had been hacked. This is why Twitter had to deal with a great deal of outcry and also why it could not explain the decision to shut down its OAuth authentication system. Hammar-Lahav explains that many of the other companies using OAuth (including Netflix, Google, Yahoo, and a host of other small companies) sent e-mails to Twitter “thanking them for taking that hit.”54 Hammar-Lahav also convinced all 30 companies involved to keep quiet for one week, and he organized an e-mail list for this group as everyone set to work to fix the problem.

Keeping the secret was difficult, and CNET actually got wind of the story early on. However, as Kirkpatrick reports, they chose not to publish the information they had “in the interest of online safety.”55 The reporter choosing not to release this information was at the Foo Camp conference along with a large number of OAuth adopters, and she may have received word of the exploit there. However, CNET did eventually force the hand of Twitter and the OAuth group. Twitter posted an announcement on its blog, acknowledging that they were attempting to fix a security gap in OAuth. The OAuth blog did the same, and it also publicly thanked Twitter for “helping to minimize premature publicity of this threat.”56 Soon after, the problem was fixed. The group developed a solution on the very last day of their one-week, self-imposed gag order. To fix the problem, the OAuth protocol was rewritten to ensure that “the redirection URI used to obtain the authorization code, is the same as the redirection URI provided when exchanging authorization code for an access token.”57 With this solution implemented, hackers could no longer stop the process in the middle, lead users to a trap site, and then make use of the token.

The emergence of this exploit and the response to it were very different from the case of the onMouseover exploit. Hammar-Lahav explains that this exploit was something that the OAuth community had missed: “This has been a solution that has been reviewed for a year and a half now, and it has been reviewed by most well-known security experts and they just missed it. Nobody ever thought of this particular security exploit.”58 As I have already argued, this is entirely unavoidable when it comes to developing software, and it is the reason I have chosen to discuss these issues in terms of hospitality. Any software or protocol must somehow deal with the predicament of hospital-
ity, with the arrival of others. Even the best web security software in the world will have to deal with exploits, but the events leading up to the patching of this security gap are striking. Consider that many of the stakeholders were present at the same conference, allowing them to meet and discuss a solution face-to-face. Also consider that the technology press and corporations were able to control when information about the exploit leaked. Whereas the onMouseover exploit was released into the wild and dealt with on the web, the OAuth exploit was addressed before it could wreak havoc on the network. The conversation that happened in the wake of the OAuth exploit’s discovery happened among a relatively tight circle of developers, companies, and security experts. Those experts implemented a solution, and the more public discussion about the exploit happened after that solution was in place.

In the case of the OAuth exploit, the conversation happened behind closed doors in order to protect the reputations and profit margins of companies. This mirrored the discussion that led to the writing of the OAuth protocol, and as we have seen when examining the controversy over signatures, that discussion was not always driven by what would make user credentials the most secure. The decision to do away with signatures in OAuth 2.0 was seen by many as a mistake, but it was a decision driven by the companies that were going to dictate whether OAuth would become a useful protocol. The only way a protocol like OAuth sees the light of day is if it is adopted by the likes of Google, Twitter, and Microsoft. This complex web of interests and programmers means that conversations about code and security take strange turns and happen outside of public view. This same type of conversation happened when the OAuth exploit emerged. The way in which this community dealt with the exploit was entirely different from the conversation about code and protocols that emerged after Kinugawa initiated his “RainbowTwtr” account and began exposing a flaw in Twitter’s parser. The OAuth exploit was (potentially) much more harmful than the onMouseover exploit, but this does not change the fact that the two emerged (and were addressed) in very different ways. Understanding these conversations, how and why they happened, and the decisions made are crucial to understanding the possibility space regulated and instituted by software and protocols.

Difficult Ethics

In her meditation on Galloway and Thacker’s exploit, Cynthia Haynes reminds us that “there is a thin line between exploit and exploitation.” Haynes focuses her attention on the deployment of IBM technology to sort people, paying special attention to a World War II era punch card’s “Hole 8,” which was punched to designate that a Nazi prisoner was Jewish. This use of compu-
tation must be seen alongside contemporary attempts to sort by way of metadata: “sorting machines became the search engines ranking and optimizing the fate of millions of Jews.” Hole 8 was an exploit, forcing us to carefully consider our consideration of the exploit as an ethical program. Haynes suggests that any theorization of the exploit must come to terms with the fact that “you cannot penetrate a system without your own version of malicious code.” Put in the terms laid out in this book, there is no ethical program that does not fall short of the Law of hospitality, smuggling in its own ethical problems. This is unavoidable, and a close analysis of any exploit makes this clear. This is why an ethics and rhetoric of the exploit is difficult, but this is also an entirely unavoidable problem.

The two exploits examined in this chapter demonstrate these difficulties, calling into question who is responsible and what the “correct” course of action is when one is confronted with a gap in the computational infrastructure. “Projecting change” through that gap, the strategy that Galloway and Thacker describe, will lead in multiple directions at once. It is difficult to plan for what follows the exploit, which provides yet another reason to question arguments that computation necessarily forecloses decision or operates “mechanically.” Recall from the introduction that Levinas is guarding against the “simple subsumption of cases under a general rule, of which a computer is capable.”

The fear is that computation cuts off the possibility of an authentic future (one that has not been programmed), an idea I return to in the conclusion of this book. But while that fear might be founded when any ethical program goes unquestioned, the examples traced out in this chapter demonstrate that any ethical program presents a complex ethical problem. We often end up with more questions than answers.

This is especially visible in the conversation that emerged after the on-Mouseover exploit. In the push to avoid malicious exploits like Kinugawa’s, users were encouraged to move away from Twitter’s web interface and toward third-party applications. Such recommendations help to initiate a significant shift in how the possibilities of digital spaces are determined and explored. Exploits provide a “teaching moment,” but that moment could easily be smoothed over in an attempt to plug the hole and move on. How might we use such situations differently? What would change if we thought of such exploits as moments when the public can be educated about the spaces in which they interact? What happens to the possibilities of exploring the boundaries, limits, and possibilities of software when exploits are addressed behind closed doors? In an environment that moves away from the web and toward apps, we can still have conversations about code among programmers and nonprogrammers alike, and the move to APIs arguably allows for spaces that are more hackable. But that hackability is clearly defined by companies such as Twitter, Google, Micro-
soft, and Facebook that determine what will be available. This is exactly what I have in mind when I argue that ethical programs have to engage with the difficult questions of hospitality. APIs are an answer to the arrival of the exploit, which pushes possibility to its limit. The exploit will often explore such possibilities regardless of utility or rationality. Assuming we are willing to expand the received notion of rhetoric as the art of the probable and to move rhetoric closer to the realm of the possible, the exploit finds a home in rhetorical theory. The exploit, as ethical program, demonstrates what is possible.

There will always have to be filters for the possibilities opened up by exploits, as the Twitter response to the onMouseover exploit shows. However, the move to APIs actually provides a much higher level of control to the likes of Facebook and Twitter (and Google) under the guise of freedom. Nonetheless, the move to APIs and apps also affords us an opportunity. While we may be moving to a world in which we have fewer opportunities to click “view source,” we can still gain some insight into the infrastructures of digital spaces. The API will filter that insight and will only allow us access to certain ways of manipulating the resources and databases of companies like Twitter. But understanding the API, protocols, and third-party applications as responses to the predicament of hospitality shows us that digital spaces constantly engage the difficulties and promises of hospitable networks. Understanding how such APIs work will require a procedurate public that takes advantage of the opportunity to understand and tinker with all of the possible means of persuasion. The task is not necessarily to ensure that everyone knows how to exploit, hack, transform, or reshape digital spaces. My argument is not that everyone should be a master programmer or that everyone should have complete access to code, though I do believe that more people learning to write code would be a good thing. Rather, what’s most important is that we work to understand how the possibilities of our digital spaces are determined and how (or whether) they can be rewritten or reenvisioned.

Who is at the table when software and protocols are written? Who is present when an exploit is discovered and patched? How does that conversation affect users and developers alike? These are the questions we land on when we consider code and software to be part of the rhetorical situation. The possibilities of a given networked environment are shaped by code and by discussions about code. What we have seen in this chapter is that exploits are rhetorical in two senses: they demonstrate what’s possible in a given space and they trigger discussion about how those spaces should operate. If the exploit is about the possible, then it is rich with ethical and rhetorical questions: What is possible? What should be possible? How do we write, code, and interact in digital spaces? Such questions can and should be addressed by scholars from multiple disciplines and by a procedurate public.